# 3DS Programming Manual: System

Version 1.6

**Nintendo Confidential**

This document contains confidential and proprietary information of Nintendo, and is protected under confidentiality agreements as well as the intellectual property laws of the United States and of other countries. No part of this document may be released, distributed, transmitted, or reproduced in any form , including by any electronic or mechanical means and by including within information storage and retrieval systems, without written permission from Nintendo.

---

©2015–2016 Nintendo. All rights reserved.

All company and product names in this document are the trademarks or registered trademarks of their respective companies.

CTR-06-0019-002-F

# 1. Introduction

This document provides 3DS application developers with an overview of CTR features, functions used, and programming procedures.

2. System provides hardware block diagrams and overviews of the various features. Refer to this chapter for an overall grasp of the whole system.

3. Memory and Nintendo 3DS Game Cards describes system memory regions and Nintendo 3DS Game Card memory regions. Refer to this chapter to learn about memory maps and proper CTR memory usage.

4. Software Configuration describes the various types of software and libraries, and how they are organized. Refer to this chapter for information about differences between the hardware and application types, and about the libraries included in the SDK.

5. Initializing Applications and Handling State Transitions describes the processes required to run an application and to properly transition through system states such as Sleep Mode. Refer to this chapter for explicit initialization and state transitioning procedures, and code examples.

6. Input From Input Devices describes how to use the various input devices in an application. Refer to this chapter for more information about using key input, accelerometer, touch panel, gyro sensor, microphone, and camera.

7. File System describes how to access media files and directories. Refer to this chapter for details on accessing files located on a Nintendo 3DS Card and its backup memory, or in SD cards.

8. Time describes time-related features, such as ticks, timers, alarms, and the real-time clock (RTC). Refer to this chapter for more information about measuring the passage of time or using the RTC.

9. Threads describes threads and classes used to synchronize multiple threads. Refer to this chapter for more information about how to create threads and how to share resources between multiple threads.

10. Sound describes how to use the libraries needed for sound playback. Refer to this chapter for more information about playing back sounds in your application.

11. System Settings describes how to access settings such as the user's name or the sound output mode. Refer to this chapter for more information about retrieving such information in your application.

12. Applets describes how to use applets from within an application. Refer to this chapter to learn how to use applets provided on the 3DS.

13. Supplemental Libraries describes how to use the libraries that provide functionality specific to the 3DS family. Refer to this chapter to learn how to access pedometer information and built-in resources.

14. Infrared Communications Between Systems describes how to use the libraries needed by the infrared communications module. Refer to this chapter to communicate between systems using infrared communications.

15. Differences Between TWL Mode and Actual TWL System describes the differences in operations of a 3DS system operating in TWL emulation mode and an actual TWL system. Refer to the cautions mentioned in this chapter when developing applications for Nintendo DS-family systems that will also run on 3DS systems.

> **Note:** For more information about the graphics features, see the separate *3DS Programming Manual: Basic Graphics* and *3DS Programming Manual: Advanced Graphics*.



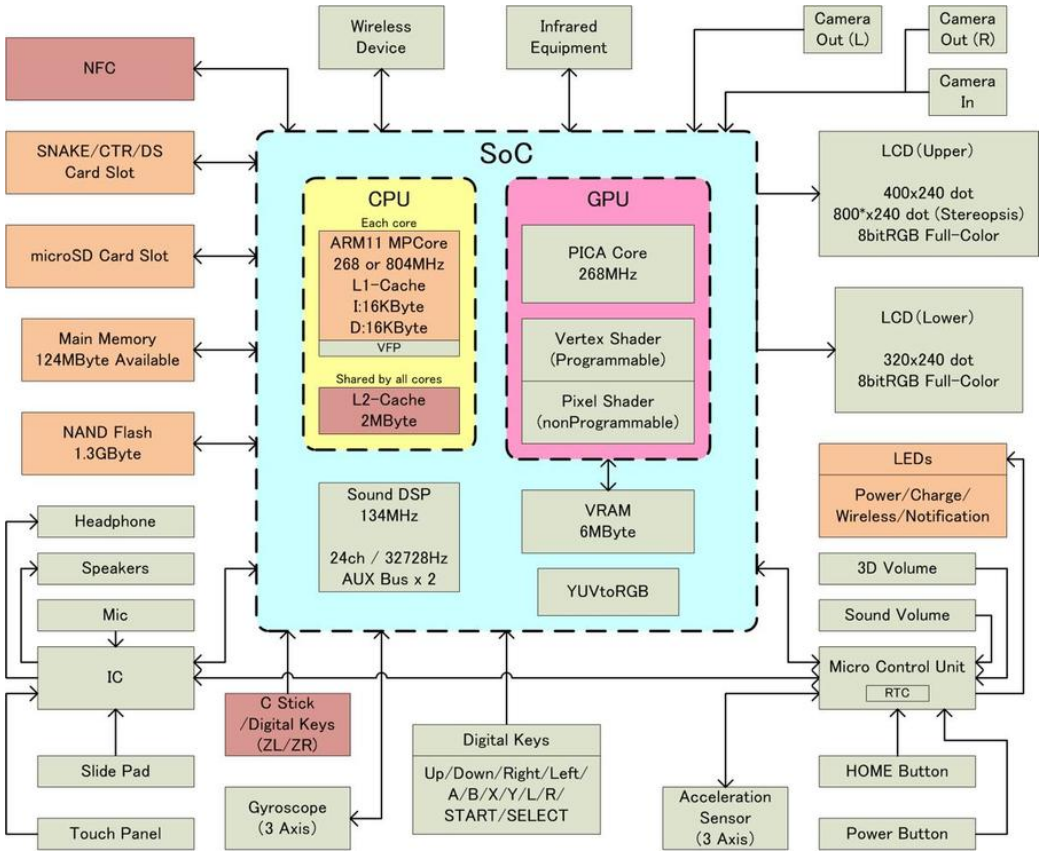
# 2. System

This section describes the system block diagrams and hardware block information for both SNAKE and CTR.

## 2.1. System Block Diagram (SNAKE)

Figure 2-1 shows an overview of the SNAKE system.

**Figure 2-1. System Block Diagram (SNAKE)**




The 800 pixels for stereoscopic display mode consist of 400 pixels each for the left and right eyes.

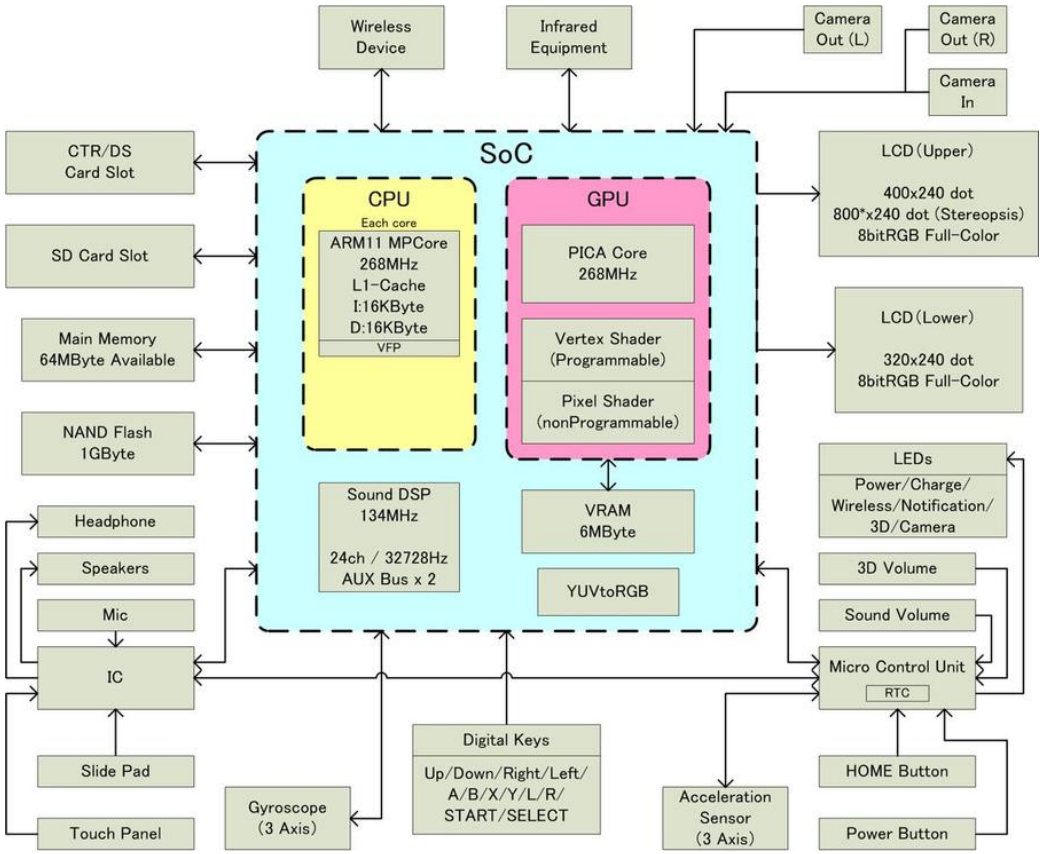
The items with a red background in the diagram are newly added specifications in SNAKE. Items with an orange background have changed from CTR.

**Note:** The LCD screen size and other specifications differ depending on the hardware variation. For information about the specification differences, see *3DS Overview – General*.

## 2.2. System Block Diagram (CTR)

Figure 2-2 shows an overview of the CTR system.

**Figure 2-2. System Block Diagram (CTR)**




The 800 pixels for stereoscopic display mode consist of 400 pixels each for the left and right eyes.

**Note:** FTR (Nintendo 2DS) has different specifications than CTR for the number of speakers and the upper screen. For information about the specification differences, see *3DS Overview – General*.

Application developers do not need to consider the differences between CTR and FTR.

## 2.3. SoC

The system-on-chip (SoC) includes the main components of the CPU, GPU, DSP, and VRAM all on one chip. The specifications of each are as follows.

### 2.3.1. CPU

This is an ARM11 MPCore processor with multiple cores and their related vector floating-point (VFP) coprocessors.

| | SNAKE | CTR |
|---|---|---|
| Processor Core | ARM11 MPCore | |
| Operating | 268 MHz or 804 MHz | 268 MHz |

| | | |
|---|---|---|
| Frequency | | |
| L1 cache | 4-way set-associative (per core) | |
| | Core 0-1:<br>16 KB for instructions, 16 KB for data<br>Core 2-3:<br>32 KB for instructions, 32 KB for data | Core 0-1:<br>16 KB for instructions, 16 KB for data |
| L2 cache | 16-way set-associative<br>(Shared by all cores. 2 MB shared command/data.) | None |
| Endianness | Little-endian | |

CTR has two CPU cores (Core 0 and Core 1). SNAKE has four CPU cores (Core 0 through 3). An application can use CPU Core 0 exclusively, while the remaining cores are used by the system. The core used exclusively by the application is called the *application core* and the cores used by the system are called the *system cores*. Each CPU core is used as follows.

| CPU Core | Usage |
|---|---|
| Core 0 | Application |
| Core 1 | System (but applications can use up to 30%) |
| Core 2 | Unused (reserved for system) |
| Core 3 | System (controls the Super-Stable 3D feature) |

**Note:** Although part of the system core can be used by the application, there are disadvantages such as a negative effect on background processes.

## 2.3.2. GPU

This includes a graphics core developed by Digital Media Professionals Inc. (DMP).

Because graphics processing is handled by the framebuffers, the BG or OBJ concepts used for the Nintendo DS do not apply.

| | |
|---|---|
| Graphics Core | PICA graphics core |
| Operating Frequency | 268 MHz |
| Architecture | Framebuffer architecture |
| Shader | Vertex (programmable), pixel (not programmable) |
| Graphics Features | OpenGL ES 1.1-based fixed pipeline + independent extensions (partially equivalent to OpenGL ES 2.0)<br>Frequently used 3D graphics processes are built into the hardware. |

## 2.3.3. VRAM

This includes two 3-MB video memory modules, which are read/write accessible only from the GPU. There is no difference in the performance of the two memory modules.

This is generally used as the location for the color buffers, depth buffers (Z buffers), and stencil buffers, and for textures and vertex buffers that are frequently used.

VRAM is independent of the main memory and cannot be written to directly by the CPU.

### 2.3.4. Sound DSP

The CTR system comes equipped with a digital signal processor (DSP). Unlike the DSP in the TWL, the CTR's DSP cannot be used for other processing.

| | |
|---|---|
| Number of Channels | 24 ch |
| Sampling Rate | Roughly 32,728 Hz (precisely, Fs = 16756991 * 16 / (32 * 256) = 32,728.49805... Hz) |
| Parameter Update Interval | Roughly 4.889 ms (precisely, T = 160 / Fs = 4.888705... ms) |
| Sampling Format | DSP ADPCM / PCM8 / PCM16 |
| Resampling | Polyphase filter / linear interpolation / disabled |
| Number of AUX Buses | 2 |

## 2.4. Main Memory

Each system has the following memory space available.

| | SNAKE | CTR |
|---|---|---|
| Main Memory | 124 MB<br>(Development hardware has 178 MB.) | 64 MB<br>(Development hardware has 96 MB.) |

Texture and vertex buffers may also be located in main memory and referenced directly from the GPU.

For more information about the memory map, see 3.1. Memory.

## 2.5. Operating Modes

The SNAKE hardware has two modes of operation: a *standard mode* that is equivalent to CTR, and an *extended mode* that is unique to SNAKE.

The following table lists the performance differences between standard mode and extended mode.

**Table 2-1. Differences Between Standard Mode and Extended Mode**

| | Standard Mode | Extended Mode |
|---|---|---|
| CPU Operating Frequency | 268 MHz | 804 MHz (3x) |
| | | |

| | | |
|---|---|---|
| CPU L2 Cache | Disabled | Enabled (2 MB) |
| Main Memory Size | 64 MB | 124 MB |

## 2.6. LCD

The system has two LCD screens.

| | | SNAKE | CTR (SPR) | FTR |
|---|---|---|---|---|
| Upper Screen | Screen size | 3.88 inches<br>XL is 4.88 inches. | 3.5 inches<br>SPR is 4.88 inches. | 3.5 inches |
| | Resolution (3D display) | 400 × 240 pixels<br>800 × 240 pixels (with one pixel for the left eye and one for the right displayed in the same size as a single pixel in normal 2D display) | | |
| | Color depth | 8-bit RGB for approximately 16.77 million colors | | |
| | LCD type | Semi-transmissive (anti-glare) | | |
| | Backlight | Built-in active backlight (controlled by turning power-saving mode on and off)<br>Brightness can be adjusted while the HOME Menu is displayed. | | |
| | 3D display | By setting the mode and designing an application to support it, an application can support autostereoscopic display (stereoscopic 3D without glasses). | | None |
| Lower Screen | Screen size | 3.33 inches<br>XL is 4.18 inches. | 3.0 inches<br>SPR is 4.18 inches. | 3.0 inches |
| | Resolution | 320 × 240 pixels | | |
| | Color depth | 8-bit RGB for approximately 16.77 million colors | | |
| | LCD type | Semi-transmissive (no anti-glare) | | |
| | Backlight | Built-in active backlight (controlled by turning power-saving mode on and off)<br>Brightness can be adjusted while the HOME Menu is displayed. | | |
| | Touch panel | Resistive film (No multi-touch. Input coordinates can be obtained in LCD pixels.) | | |

**Warning:** Stereoscopic display does not work when the system is held vertically (with the longer side of the LCD screens vertical).

**Note:** For more information about stereoscopic display, see the *3DS Programming Manual: Advanced Graphics*.

## 2.7. Memory Devices

## 2.7.1. Game Card Slot

The Game Card slot accepts the following Game Cards.

| | 3DS Only | DS Only<br>DSi-Compatible<br>DSi Only |
|---|---|---|
| SNAKE | ○ | ○ |
| CTR | ○ | ○ |

**Note:** It does not have a Game Pak (GBA) slot.

### 2.7.1.1. Nintendo 3DS Game Cards

Nintendo 3DS Game Cards can be accessed at higher speeds than TWL/NITRO Cards, and security features have also been revamped.

Two types of cards are available: CARD1 and CARD2. Current specifications support a CARD1 ROM capacity of up to 4 GB (32 gigabits). A backup memory capacity of either 128 KB (1 megabit) or 512 KB (4 megabits) is supported. CARD2 provides a total ROM and backup memory capacity of up to 2 GB (16 gigabits).

The system uses some of the capacity of both the ROM and backup memory. For more information, see 3.2. Nintendo 3DS Game Cards and 7.3. Save Data.

## 2.7.2. System NAND Memory

Each system has the following amount of internal NAND memory. System NAND memory is used for saving data such as preinstalled application data.

| | SNAKE | CTR |
|---|---|---|
| Capacity | 1.3 GB | 1 GB |

## 2.7.3. SD/microSD Card Slot

Each system includes a microSD or SD card slot that supports the following media types.

The system provides a means of accessing some files using dedicated libraries (for example, SoundDB and ImageDB). Applications (specifically, downloadable applications other than Nintendo DSiWare) can be started directly from an SD card.

| | SNAKE | CTR |
|---|---|---|
| Supported Media | microSD Memory Card<br>microSDHC Memory Card<br>(microSDXC Memory Cards are not supported.) | SD Memory Card<br>SDHC Memory Card<br>(SDXC Memory Cards are not supported.) |

# 2.8. Input Devices

## 2.8.1. Key Input

| | SNAKE | CTR SPR | FTR | Comments |
|---|---|---|---|---|
| +Control Pad | Yes | | | |
| A, B, X, and Y Buttons | Yes | | | When the X Button on SNAKE is pressed with considerable force, downward movement might be detected on the C Stick. To avoid problems, we recommend the following workarounds in applications that use the C Stick.<br>• Do not assign press-and-hold operations to the X Button.<br>• Do not read values from the C Stick in scenarios where the X Button is held down.<br>• Adjust the sensitivity of the C Stick.<br>More force is required to reproduce this issue on the development hardware and the new Nintendo 3DS than on the new Nintendo 3DS XL. |
| L/R Buttons | Yes | | | |
| ZL/ZR Buttons | Yes | Available when Circle Pad Pro is connected. | None | The ZL and ZR Buttons on SNAKE are compatible with the ZL and ZR Buttons on the Nintendo 3DS Circle Pad Pro. Applications can handle them as if SNAKE were a CTR system that is always equipped with the Circle Pad Pro. |
| Circle Pad | Yes | | | |
| C Stick/ Right Circle Pad | Yes | Available when Circle Pad Pro is connected. | None | The C Stick on SNAKE is compatible with the Right Circle Pad on the Nintendo 3DS Circle Pad Pro. Applications can handle them as if SNAKE were a CTR system that is always equipped with the Circle Pad Pro. |
| START or SELECT | Yes | | | SELECT has been kept for backward compatibility with Nintendo DS software, and any SELECT press is interpreted by 3DS applications as identical to a START press. |
| HOME Button | Yes | | | Activates the HOME Menu.<br>This input cannot be used by the application for gameplay purposes. |
| POWER Button | Yes | | | Controls power features. To prevent unintended operation of the button, the button does not react to momentary presses. Holding it down for a set period of time begins forced shutdown.<br>On SNAKE hardware, the POWER Button can be pressed while the system is closed.<br>This input cannot be used by the application for gameplay purposes. |
| Wireless switch | None | Yes | None | On SNAKE and FTR, wireless communication is enabled and disabled from the HOME Menu.<br>This input cannot be used by the application for gameplay purposes. |

| | | | |
|---|---|---|---|
| Open/close detection sensor/ Sleep Switch | Open/close detection sensor | Sleep Switch | The open/close detection switch detects whether the system is open or closed. FTR cannot be closed, so it includes a sleep switch "slider" instead of an open/close detection switch.<br>This input cannot be used by the application for gameplay purposes. |
| Sound volume | Yes | | Equipped with a volume slider.<br>This input cannot be used by the application for gameplay purposes. |
| 3D Depth Slider | Yes | None | Adjusts parallax for stereoscopic display.<br>This input cannot be used by the application for gameplay purposes. |

## 2.8.2. Accelerometer

The CTR system includes an accelerometer in the base of the system that measures acceleration on each of the three axes. This can be used by applications. The accelerometer range is approximately 1.8 G in both directions on each axis, with noise when stationary of up to ±0.02 G, sensitivity of approximately 0.002 G, and a sampling rate of 100 Hz (theoretical value for the device).

## 2.8.3. Touch Panel

The CTR system has a touch panel over the lower LCD, just like the Nintendo DS/DSi.

The touch panel uses resistive film technology for single-point touch functionality similar to the Nintendo DSi. As with previous systems, input coordinates can be obtained in terms of LCD pixels.

## 2.8.4. Microphone

The monaural microphone is located to the side of the group of buttons at the bottom of the lower screen. Performance is equivalent to the microphone on the Nintendo DSi. As with the Nintendo DSi, the microphone gain can be adjusted from 10.5 dB to 70.0 dB in increments of 0.5 dB.

Sensitivity variations between individual microphones are within 0.5 dB (1.06x).

## 2.8.5. Cameras

The CTR system has one inner camera and two outer cameras (left and right), with the same specifications as the ones used on the Nintendo DSi system. You can either use both outer cameras at the same time or the left outer camera and the inner camera at the same time, but it is not possible to use all three cameras at the same time.

The main specifications of the cameras are as follows.

| | |
|---|---|
| Aperture | F2.8 (fixed). |
| Angle of view | See the following table (when photographing at maximum resolution). |

| | |
|---|---|
| Photographable range | 20 cm to infinity. (Pan focus. Not equipped with a macro switch.) |
| Maximum resolution | VGA |
| Maximum frame rate | 30 fps (fps: frames per second) |
| Output format | YCrYCb (can also output to RGBA8888, RGB888, RGB565, and RGBA5551 formats by use of the separate YUV to RGB circuit). |

| | Minimum | Average | Maximum |
|---|---|---|---|
| Diagonal | 63.0° | 66.0° | 69.0° |
| Horizontal | 52.2° | 54.9° | 57.6° |
| Vertical | 40.4° | 42.6° | 44.8° |

### 2.8.6. Gyro Sensor

The CTR system is equipped with a triaxial gyroscopic sensor in the bottom half of the system that can detect when the system is tilted and the speed of rotation. The sensor can measure up to ±1,800 degrees per second (DPS), with noise when stationary of up to ±2.28 DPS, sensitivity of 0.07 DPS, and a sampling rate of 100 Hz.

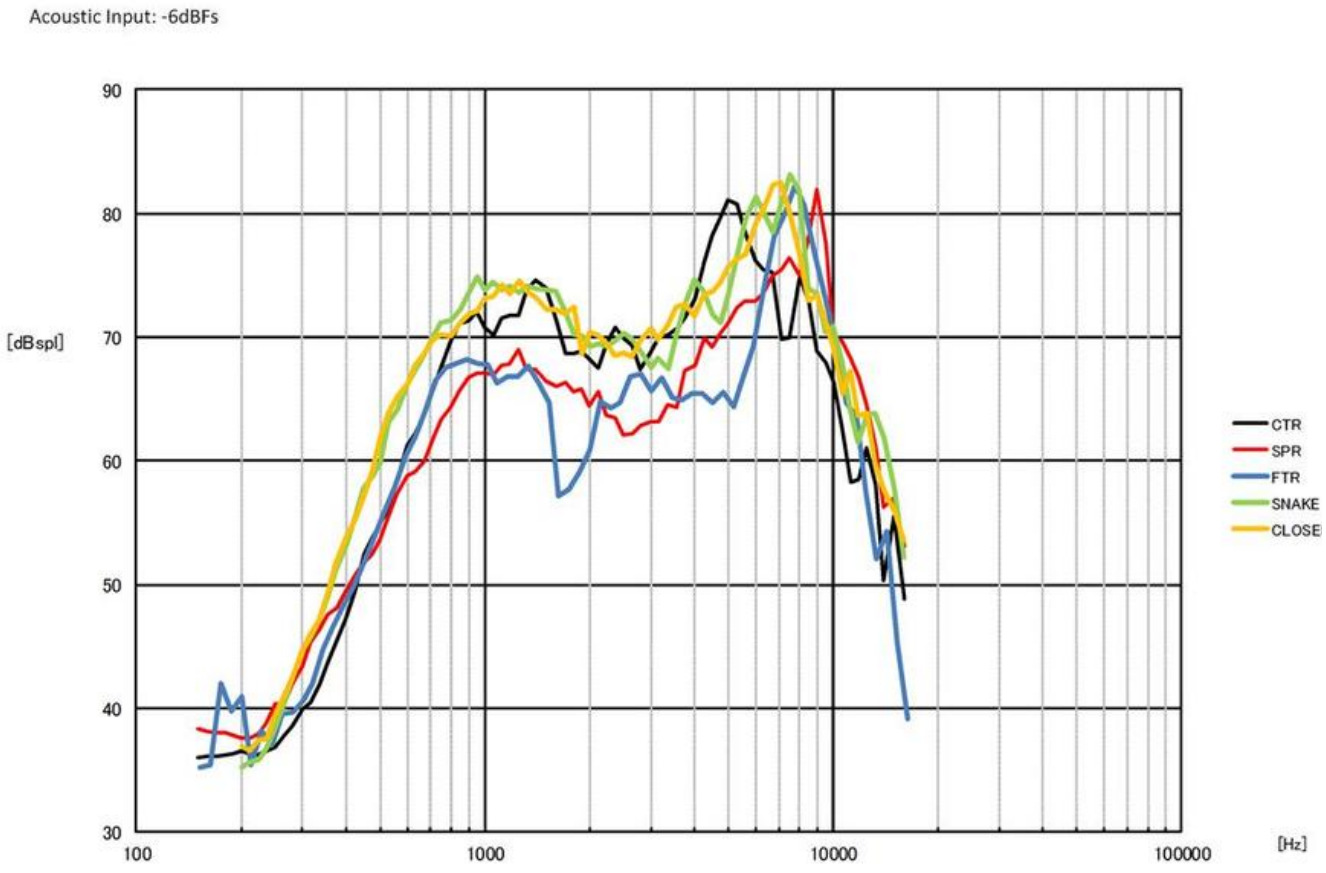
## 2.9. Output Devices

### 2.9.1. Speakers

CTR, SPR, SNAKE, and CLOSER have stereo speakers located to the left and right of the upper screen. FTR has a single monaural speaker located to the upper-left of the upper screen.

The following figure shows the sound pressure frequency characteristics of the CTR, SPR, FTR, SNAKE, and CLOSER speakers.

**Figure 2-3. Sound-pressure Frequency Characteristics of the CTR (SPR/FTR/SNAKE/CLOSER) System Speakers**




## 2.9.2. Audio Jack

The system comes equipped with a stereo output mini-jack. Unlike the Nintendo DS/DSi, there is no microphone jack.

Headsets that include a microphone are not supported.

| | SNAKE | CTR | SPR | FTR |
|---|---|---|---|---|
| Audio Jack location | Front and center of the lower half of the system | | Front left of lower half of system | |

## 2.9.3. LEDs

The system comes with LEDs to indicate the camera state, battery level, charging status, wireless status, 3D display, and notification status.

Each platform has the following LEDs.

| | SNAKE | CTR | SPR | FTR |
|---|---|---|---|---|
| Camera status | None | Yes | Yes | None |
| Battery | Yes | Yes | Yes | Yes |
| Charging | Yes | Yes | Yes | Yes |

| | | | | |
|---|---|---|---|---|
| Wireless | Yes | Yes | Yes | Yes |
| 3D Display | None | Yes | None | None |
| Notifications | Yes | Yes | Yes | Yes |

# 2.10. Communication Devices

## 2.10.1. Wireless Communication Module

The CTR system has a wireless communication module that transmits in the 2.4 GHz band.

Communications features can be broadly classified as foreground, such as when an application explicitly uses communications features, and background, such as when the system is engaging in automatic communications.

For more information about wireless communication, see the *3DS Programming Manual: Wireless Communication*.

- Foreground communication
  - Infrastructure communication
  - Local communication
  - Download Play

- Background communication
  - StreetPass
  - Download tasks
  - Presence features

## 2.10.2. Infrared Communication Device

The CTR system comes with an infrared transceiver and an infrared communication device.

## 2.10.3. Near Field Communication (NFC)

A contactless near field communication (NFC) antenna is located below the lower screen panel for SNAKE.

## 2.11. Other

### 2.11.1. Real-Time Clock (RTC)

This keeps and measures time.

### 2.11.2. Compatibility With the Nintendo DSi

The CTR system maintains compatibility with the Nintendo DSi and supports DS Download Play.



# 3. Memory and Nintendo 3DS Game Cards

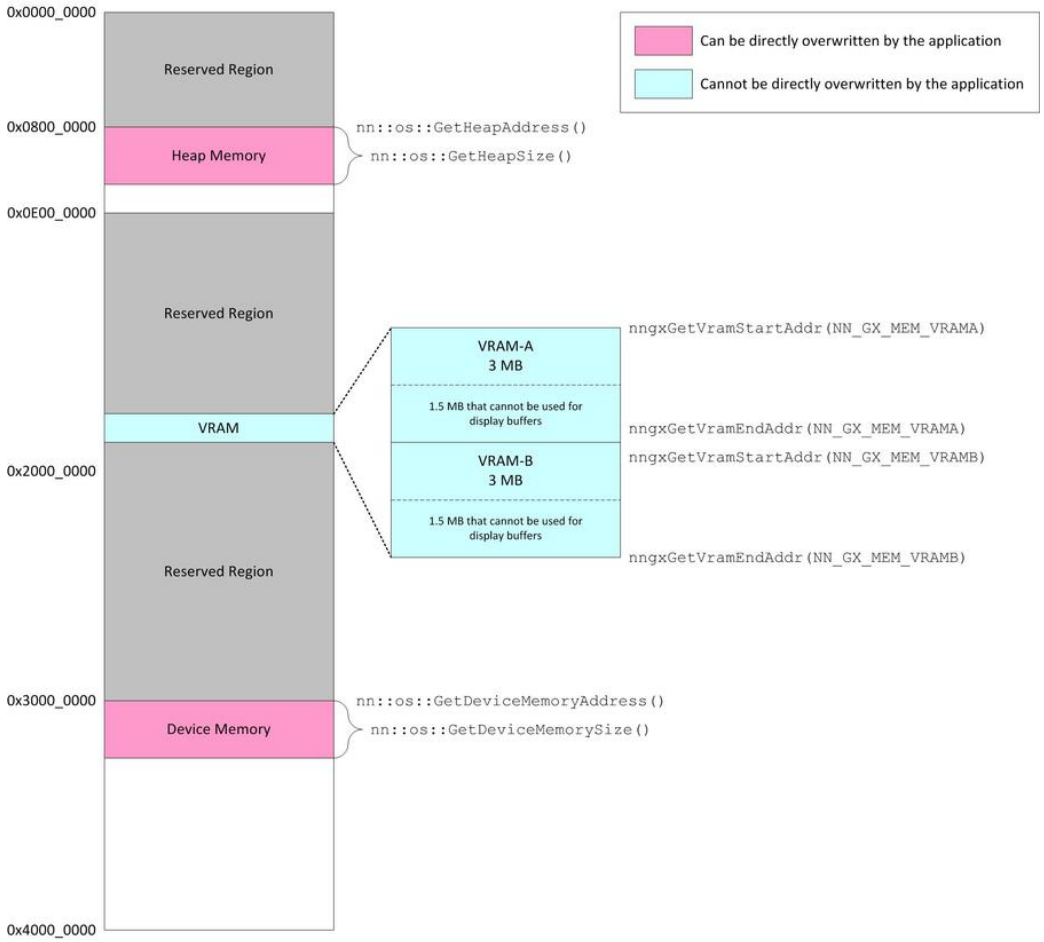
This chapter describes memory regions that can be accessed by applications and Nintendo 3DS Game Cards written to by card-based software.

## 3.1. Memory

### 3.1.1. Memory Map

The following figure shows a memory map as seen by applications. This memory map is for release versions.

**Figure 3-1. Memory Map (Release Versions)**




Memory access from applications is carried out using virtual addresses.

For security reasons, it is not possible to execute data arrays loaded into memory other than the region where the program is loaded (such as heap memory or device memory).

Memory above address `0x4000_0000` is reserved.

## 3.1.2. Device Memory

Device memory is a region in memory for which the operating system guarantees address integrity when it is accessed by the GPU or other peripheral device. Some of the buffers accessed by the GPU used for graphics and the DSP used for sound must be allocated from this device memory. Conversely, some buffers cannot be allocated from device memory. Generally, those buffers must be 4096-byte aligned and have sizes that are multiples of 4096 bytes.

You can use the `nn::os::SetDeviceMemorySize()` function to specify the size of the memory region to allocate as device memory. Specify the size as a multiple of `nn::os::DEVICE_MEMORY_UNITSIZE` (4096 bytes). On CTR, you can allocate up to 64 MB for the main program, heap memory and device memory (96 MB on development hardware). On SNAKE, you can allocate up to 124 MB (178 MB on development hardware). You can call this function again to change the size of the memory region to allocate, but if you do you must change the size by a multiple of 1 MB (1,048,576 bytes). When changing to size 0 at this time, there are no issues if the size before the change is a multiple of 1 MB. Similarly, when changing from size 0, there are no issues if the size before the change is a multiple of 1 MB. However, it has to be made a multiple of 4096 bytes.

There is no guarantee that the address of the memory region allocated will be the same for every

call. After specifying the size of the region, first use the `nn::os::GetDeviceMemoryAddress` function to get the starting address of the region and then access the device memory. Use the `nn::os::GetDeviceMemorySize` function to get the size of the currently allocated memory.

**Table 3-1. Types and Starting Address Alignment for Buffers Allocated From Device Memory**

| Device | Buffer Type | Alignment | Comments |
|---|---|---|---|
| GPU | Texture image | 128 bytes | |
| | Vertex buffers | 1 to 4 bytes | Changes depending on vertex attributes. |
| | Display buffer | 16 bytes | |
| DSP | Sound source data | 32 bytes | The size of the whole region storing sound source data must be a multiple of 32 bytes. |
| CAMERA | Receive buffer | 64 bytes (recommended) | If you use a buffer allocated in a space other than device memory, the SDK stops on a panic. |
| Y2R | Image data receive/send buffer | 64 bytes (recommended) | If you use a buffer allocated in a space other than device memory, the SDK stops on a panic. |

Buffers accessed by the GPU may also be allocated from VRAM.

**Table 3-2. Buffers That Cannot Be Allocated From Device Memory**

| Buffer | Comments |
|---|---|
| Buffer storing microphone sampling results | Must be 4096-byte aligned, and its size must be a multiple of 4096 bytes. |
| UDS library's receive buffer | Must be 4096-byte aligned, and its size must be a multiple of 4096 bytes. |
| Socket library's receive buffer | Must be 4096-byte aligned, and its size must be a multiple of 4096 bytes. |
| HTTP library's working buffer | Must be 4096-byte aligned, and its size must be a multiple of 4096 bytes. |
| DLP library's working buffer | Must be 4096-byte aligned, and its size must be a multiple of 4096 bytes. |
| ACT library communication buffer | Must be 4096-byte aligned, and its size must be a multiple of 4096 bytes. |

## 3.1.3. VRAM

VRAM consists of the two regions VRAM-A and VRAM-B, each 3 MB, for a total of 6 MB.

Applications must manage memory regions used by the GX (graphics) library, such as by allocating color buffers. (Note that when reading VRAM on the CPU, the data is not guaranteed to be accurate.) To allocate memory in VRAM, use the `nngxGetVramStartAddr`, `nngxGetVramEndAddr`, and `nngxGetVramSize()` functions to get the start and end addresses and size of VRAM to manage the location of the requested memory regions.

For more information about the GX library and how to manage memory used by libraries, see 5.5. Initializing the GX Library (in this manual) and the *3DS Programming Manual: Basic Graphics*.

## 3.1.4. Heap Memory

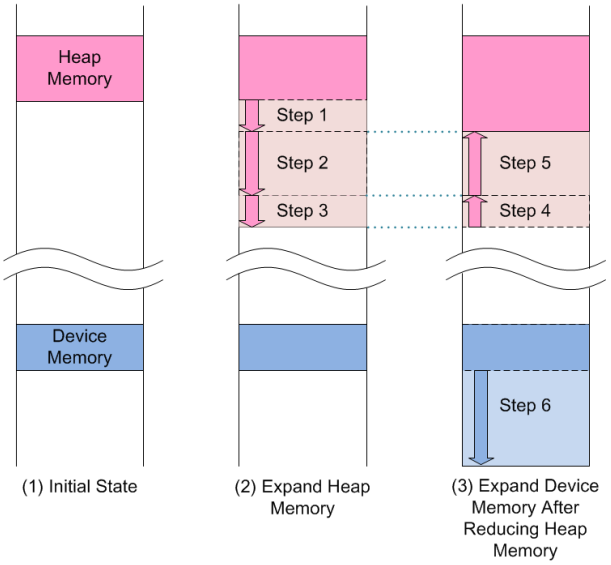
This memory is freely usable by applications, such as for allocating file read and write buffers. Use the `nn::os::SetHeapSize` function to specify the size of heap memory to allocate. The size must be a multiple of `nn::os::HEAP_UNITSIZE` (4096 bytes) On CTR, the total size of main program meny, heap memory, and device memory must be no greater than 64 MB (96 MB for development units). On SNAKE, the heap can use up to 96 MB and the total size of main program memory, heap memory, and device memory must be no greater than 124 MB (178 MB for development hardware). Call this function again to change the size of the memory region to allocate.

When expanding and reducing the size of the heap memory region according to the usage status, and resizing the device memory region accordingly, take note of the following points when managing the heap memory region.

- When reducing the heap memory region after it was expanded, reduce it by the same amount it was incremented.
- The heap memory region can be divided more than once and expanded. In such cases, reduce its size in stages, so that it returns to the size prior to expansion.

If this condition is violated, it may not be guaranteed that the unassigned region can be secured as device memory.

**Figure 3-2. Example of Expanding Device Memory After Expanding or Reducing Heap Memory**




As shown in Figure 3-2, when reducing the heap memory region and expanding the device memory region from a state where the heap memory region was expanded in stages (Step 1, Step 2, Step 3), to return the heap memory region to the size it was prior to expansion, first reduce it by the same size it was incremented by in Step 3 (Step 4), and then reduce it by the same size it was increased by in Step 2 (Step 5).

**Warning:** Wherever possible, we recommend maintaining the total size required by the heap memory region and not resizing it. Perform these steps only when a resize is absolutely required. However, resizing in small units can lead to a considerable drop in performance when accessing the memory.

There is no guarantee that the address of the memory region allocated will be the same for every call. After specifying the size, use the `nn::os::GetHeapAddr()` function to get the starting address of the region. Use the `nn::os::GetHeapSize()` function to get the size of the currently allocated heap.

## 3.2. Nintendo 3DS Game Cards

There are two types of Nintendo 3DS Game Cards: CARD1 and CARD2. The card most appropriate for use is determined by the game title's specifications.

**Warning:** As a rule, use CARD1 for game titles with a backup memory capacity of 512 KB, and use CARD2 for those with a capacity of 1 MB or more.

### 3.2.1. CARD1

CARD1 cards come equipped with both read-only memory (ROM) and backup memory (rewritable non-volatile memory).

#### 3.2.1.1. ROM

The CARD1 ROM transfer speed varies depending on the version of the SDK in use and the size of the data being transferred. The actual speed may also vary slightly due to differences in individual cards, so do not design applications that depend on transfer speeds.

There are six different ROM capacities, ranging from 128 MB to 4 GB.

**Table 3-3. ROM Capacity (CARD1)**

| ROM Capacity | Transfer Speed | Usable Space Size by Market | | |
|---|---|---|---|---|
| | | Japan, North America, Korea | Europe | China, Taiwan |
| 128 MB | (See Note) | 93.5 MB (98,041,856 bytes) | 91.5 MB (95,944,704 bytes) | 61.5 MB (64,487,424 bytes) |
| 256 MB | | 219.0 MB (229,638,144 bytes) | 217.0 MB (227,540,992 bytes) | 187.0 MB (196,083,712 bytes) |
| 512 MB | | 470.0 MB (492,830,720 bytes) | 468.0 MB (490,733,568 bytes) | 438.0 MB (459,276,288 bytes) |
| 1 GB | | 921.5 MB (966,262,784 bytes) | 919.5 MB (964,165,632 bytes) | 889.5 MB (932,708,352 bytes) |
| 2 GB | | 1875.5 MB | 1873.5 MB (1,964,507,136 | 1843.5 MB (1,933,049,856 |

| | | | | |
|---|---|---|---|---|
| | | (1,966,604,288 bytes) | bytes) | bytes) |
| 4 GB | | 3783.0 MB (3,966,763,008 bytes) | 3781.0 MB (3,964,665,856 bytes) | 3751.0 MB (3,933,208,576 bytes) |

Transfer speeds depend on the version of the SDK and the size of the data. For more information, see *3DS Performance Tips*.

### 3.2.1.2. Backup Memory

The transfer speed for backup memory in CARD1 cards depends on the version of the SDK in use and the size of the data being transferred. The actual speed may also vary slightly due to differences in individual cards, so do not design applications that depend on transfer speeds. See the *CTR Guidelines* and follow its standards for the number of write operations.

There are two different backup memory capacities, either 128 KB or 512 KB. Use a CARD2 card if you need 1 MB or more of backup memory.

The content of backup memory at time of shipment is guaranteed to be padded with `0xFF` throughout the entire memory region. Pad all of backup memory with `0xFF` when emulating the memory state at time of shipment.

You can enable automatic redundancy of the backup memory region used for save data files. However, when automatic redundancy is enabled, the capacity available for saving files is roughly 40% of the total capacity.

**Table 3-4. Backup Memory Capacity (CARD1)**

| Backup Memory Capacity | Transfer Speed | Usable Region Size When Automatic Redundancy Is Enabled |
|---|---|---|
| 128 KB | (See Note) | 50 KB or less |
| 512 KB | | 239 KB or less |

Transfer speeds depend on the version of the SDK and the size of the data. For more information, see *3DS Performance Tips*.

## 3.2.2. CARD2

CARD2 cards come equipped with writable memory that has features of both ROM and backup memory.

### 3.2.2.1. Writable Memory

The transfer speed for writable memory in CARD2 cards depends on the version of the SDK used and the size of the data involved. The actual speed may vary slightly due to differences in individual cards, so do not design applications that depend on transfer speeds. See the *CTR*

*Guidelines* and follow its standards for the number of write operations.

With writable memory, you can change the boundaries of the ROM (read-only) and backup (rewritable) regions in 1-MB increments.

You can use the library to enable automatic redundancy for data in backup memory, just as with CARD1 backup memory. Calculate the usable capacity for saving files in a similar manner. The maximum file size that you can specify as backup space is half of the writable memory capacity. There are three sizes of writable memory capacity: 512 MB, 1 GB, and 2 GB.

**Table 3-5. Writable Memory Capacity (CARD2)**

| Writable Memory Capacity | Transfer Speed | Usable Space Size by Market | | |
|---|---|---|---|---|
| | | Japan, North America, Korea | Europe | China, Taiwan |
| 512 MB | (See Note) | 412.5 MB (432,537,600 bytes) | 410.5 MB (430,440,448 bytes) | 412.5 MB (432,537,600 bytes) |
| 1 GB | | 889.5 MB (932,708,352 bytes) | 887.5 MB (930,61,200 bytes) | 889.5 MB (932,708,352 bytes) |
| 2 GB | | 1843.5 MB (1,933,049,856 bytes) | 1841.5 MB (1,930,952,704 bytes) | 1843.5 MB (1,933,049,856 bytes) |

Transfer speeds depend on the version of the SDK and the size of the data. For more information, see *3DS Performance Tips*.



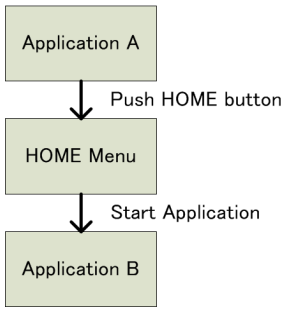
# 4. Software Configuration

This chapter describes the software configuration of the 3DS system.

## 4.1. Applications

To switch from one application to another, you must start the second application using the HOME Menu.

**Figure 4-1. Application Switching**




## 4.2. Standard and Extended Applications

The SNAKE hardware offers two operating modes with different CPU performance: *standard mode* and *extended mode*. For more information, see 2.5. Operating Modes. CTR is handled as a platform that only supports the standard mode.

Consequently, two application types are available: *standard applications* that run in standard mode even on SNAKE, and *extended applications* that run in extended mode on SNAKE.

**Table 4-1. Application Types and Operating Modes on Each Platform**

| Application Type | SNAKE | CTR | Comments |
|---|---|---|---|
| Standard application | Standard mode | Standard mode | Includes applications built with previous SDKs. |
| Extended application | Extended mode | Standard mode | |

### 4.2.1. Creating a Standard Application

If you are using the CTR-SDK build system, comment out the following line in the OMakefile if it exists: `"DESCRIPTOR = $(CTRSDK_ROOT)/resources/specfiles/ExtApplication.desc"`.

If you are using VSI-CTR (requires VSI-CTR Platform 3.0.0 or later), in **Configuration Properties** > **General**, change **Create Extended Application** to **No**.

### 4.2.2. Creating an Extended Application

If you are using the CTR-SDK build system, add the following line to the OMakefile: `"DESCRIPTOR = $(CTRSDK_ROOT)/resources/specfiles/ExtApplication.desc"`.

If you are using VSI-CTR (requires VSI-CTR Platform 3.0.0 or later), in **Configuration Properties** > **General**, change **Create Extended Application** to **Yes**.

### 4.2.3. Things to Note When Using Extended Mode

The L2 cache is enabled in extended mode. However, the performance of functions such as `nngxUpdaterBuffer` might decrease due to the large capacity of the L2 cache.

To avoid this performance drop, variants of some API functions are provided for specific purposes in extended mode. Be sure to use the right function when handling graphics in your application while running in extended mode.

## 4.2.4. When the Operating Mode Changes

On SNAKE, the operating mode changes when `nn::applet::Enable` is called. Consequently, even standard applications run in extended mode from the time the 3DS logo is displayed until control is passed to the application and `nn::applet::Enable` is called.

The system always runs in extended mode while the HOME Menu is displayed, regardless of whether an application is running. When a standard application is suspended by the HOME Menu, the system changes from standard mode to extended mode, and then back to standard mode when control returns to the application.

System applets such as Game Notes and Internet Browser also suspend the application and switch the operating mode to extended mode in the same manner as the HOME Menu.

**Table 4-2. Application Types and Timing of Operating Mode Change**

| Application Type | Logo Display | nn::applet::Enable | Application | HOME Menu | Library Applet | System Applets |
|---|---|---|---|---|---|---|
| Extended application | Extended mode | Extended mode | Extended mode | Extended mode | Extended mode | Extended mode |
| Standard application | Extended mode | Standard mode | Standard mode | Extended mode | Standard mode | Extended mode |

## 4.2.5. Changing Application Behavior on SNAKE and CTR

With the addition of SNAKE to the 3DS family, in some cases it may be necessary to modify the behavior of an application depending on the type of hardware it is running on.

An application can determine whether it is running on SNAKE hardware by calling the `nn::os::IsRunOnSnake()` function. This allows hardware-specific implementations, such as omitting the Circle Pad Pro connection check on SNAKE hardware, which has the equivalent input devices built into the system.

An application created as an extended application can determine whether it is running in extended mode on SNAKE by calling the `nn::os::IsRunningAsExtApplication()` function. Note, however, that this function has a high execution cost and cannot be called inside `nninitStartUp`.

The `nn::os::IsRunOnSnake()` and `nn::os::IsRunningAsExtApplication()` functions return the following values for the various combinations of hardware and application type.

**Table 4-3. Return Values for Hardware and Application Combinations**

| Hardware | Application | IsRunOnSnake() | IsRunningAsExtApplication() |
|---|---|---|---|
| SNAKE | Extended application | `true` | `true` |
| | Standard application | `true` | `false` |

| | | | |
|---|---|---|---|
| CTR | Extended application | `false` | `false` |
| | Standard application | `false` | `false` |

**Warning:** Always use the `nn::os::IsRunningAsExtApplication()` function to determine whether the application can use the enhanced features in extended mode (such as additional memory and CPU performance). The cache manipulation functions that are added when the L2 cache is enabled, however, can be used even in standard mode.

Use the `nn::os::IsRunOnSnake()` function to determine whether any other devices with different specifications on SNAKE and CTR can be used.

### 4.2.6. SNAKE-Only Titles

For applications that only operate on SNAKE, set the `SNAKEOnly` item in the BSF file to `True`. SNAKE-only titles do not run on CTR, so you do not need to consider operations on CTR.

**Note:** For more information about BSF files, see the reference manual for the CTR-SDK tool `ctr_makebanner`.

## 4.3. What the SDK Provides

The 3DS SDK includes not only libraries for handling device input such as key presses, but also provides applets and libraries for using features such as the HOME Menu and SpotPass.

### 4.3.1. Applets

Much like the HOME Menu, applets provide features for specific purposes that applications can use. Using applet features can help reduce the cost of developing applications.

**Warning:** Only applets provided by Nintendo can be used. Developers cannot create their own applets.

### 4.3.2. Libraries

CTR-SDK includes the libraries needed to use the hardware. Each library has its own namespace derived from the library name and comprises multiple classes and member functions.

Libraries are written mainly in C++, with wrapper functions written in C. For more information about the C-language wrapper functions, see the *CTR-SDK API Reference*. The following descriptions all

use C++ for function names and other code.

**Table 4-4. Library List**

| Library Name | Namespace | Description |
|---|---|---|
| OS | `nn::os` | A collection of classes for memory allocation, mutual exclusion, ticks, alarms, threads, and other operating system-related features. |
| RO | `nn::ro` | Provides DLL features. |
| APPLET | `nn::applet` | Supports starting applets, transitioning the system to sleep when the system is closed, and other functionality. |
| FS | `nn::fs` | Used for accessing files on various media. |
| CX | `nn::cx` | Handles data compression and decompression. |
| MATH | `nn::math` | A collection of mathematical and numeric functions. |
| CRYPTO | `nn::crypto` | Handles encryption. |
| FND | `nn::fnd` | A collection of heap, time, and other fundamental classes. |
| FONT | `nn::font` | Handles character drawing using font data. |
| HID | `nn::hid` | Handles input from the digital buttons, Circle Pad, touch panel, accelerometer, gyro sensor, and debug pad. |
| PTM | `nn::ptm` | Controls system power and alarms. |
| GX | `nn::gx` | Handles GPU and LCD control. Uses the `gl()` and `nngx()` functions for 3D graphics rendering. |
| GD | `nn::gd` | A lighter-weight version of the 3D graphics rendering functions in the GX library. |
| GR | `nn::gr` | Supports direct generation of 3D graphics commands. |
| MIC | `nn::mic` | Handles automatic microphone sampling. |
| CAMERA | `nn::camera` | Handles image capture using a camera. |
| Y2R | `nn::y2r` | Handles conversion from YUV to RGB format using the YUVtoRGB circuit. |
| QTM | `nn::qtm` | Handles face tracking. Available only on SNAKE. |
| DSP | `nn::dsp` | Allows use of DSP for sound playback |
| SND | `nn::snd` | Handles local sound playback. |
| AACDEC | `nn::aacdec` | Handles AAC data decoding. |
| AACENC | `nn::aacenc` | Handles AAC data encoding. |
| NDM | `nn::ndm` | Controls the daemon carrying out network processes. |
| UDS | `nn::uds` | Handles wireless communications using the wireless communications module. |
| RDT | `nn::rdt` | Allows secure data communication using the UDS library. |
| CEC | `nn::cec` | Handles StreetPass [Chance Encounter Communication (CEC)] settings and other features. |
| DLP | `nn::dlp` | Supports Nintendo 3DS Download Play. |
| AC | `nn::ac` | Handles automatic connections for infrastructure communication. |
| BOSS | `nn::boss` | Handles download task registration. |
| FRIENDS | `nn::friends` | Supports access to friend information. |
| NEWS | `nn::news` | Posts notifications. |
| ACT | `nn::act` | Gets information registered to the account system and authenticates it. |
| EC | `nn::ec` | For using the EC features. |
| | | |

| | | |
|---|---|---|
| IR | `nn::ir` | For using infrared communication between systems. |
| NFP | `nn::nfp` | Allows linking between the application and branded character products called amiibo™ figures. |
| ULCD | `nn::ulcd` | Calculates the left- and right-eye camera matrices used in stereoscopic display. |
| JPEG | `nn::jpeg` | Handles encoding to and decoding from JPEG format. |
| TPL | `nn::tpl` | Handles texture collection TPL files. |
| CFG | `nn::cfg` | Gets information handled by System Settings. |
| NGC | `nn::ngc` | Checks for words that are in the profanity list. |
| UBL | `nn::ubl` | Manages the blocked-user list.. |
| PL | `nn::pl` | Used for features unique to the 3DS (such as the pedometer). |
| UTIL | `nn::util` | Utility functions. |
| ENC | `nn::enc` | Handles encoding conversions. |
| ERR | `nn::err` | Error handling. |
| ERREULA | `nn::erreula` | For using the Error/EULA applet. |
| SWKBD | `nn::swkbd` | For using the software keyboard applet. |
| PHTSEL | `nn::phtsel` | For using the photo selection applet. |
| VOICESEL | `nn::voicesel` | For using the sound selection applet. |
| EXTRAPAD | `nn::extrapad` | For using the Circle Pad Pro calibration applet. |
| WEBBRS | `nn::webbrs` | For using the Internet browser. |
| OLV | `nn::olv` | For using the Miiverse application and the Post app. |
| SOCKET | `nn::socket` | (Debugging Only) Handles socket communication. |
| SSL | `nn::ssl` | (Debugging Only) Handles SSL communication. |
| HTTP | `nn::http` | (Debugging Only) Handles HTTP communication. |
| DBG | `nn::dbg` | Assists in debugging. |
| HIO | `nn::hio` | (Debugging Only) For using Host IO. |
| MIDI | `nn::midi` | (Debugging Only) For using MIDI. |

Library configurations and names may be changed in the future.

## 4.4. Error Handling

When the 3DS system is updated, the libraries are also updated. As a result, functions could return values that were not defined at the time the application was developed. To handle such cases, you must use the `nn::Result::IsSuccess()` or `nn::Result::IsFailure()` function to determine whether a process has succeeded.

Nintendo does not recommend error handling that only uses value matching to determine errors rather than using these functions, because doing so could cause the application to erroneously assume success or failure when an unexpected error occurs.

**Note:** When errors that differ from the symptoms in the reference occur only on a particular device, a hardware problem may be occurring.

# 5. Initializing Applications and Handling State Transitions

This chapter describes what an application must initialize when it is started, and how to handle state transitions, such as to Sleep Mode.

## 5.1. Initialization Prior to Calling the Entry Function

For more information about the processes performed prior to calling the application's entry function, see the *System Programming Guide* included in the SDK.

## 5.2. Entry Function

An application's entry function is defined by the `nnMain()` function.

The entry function must first initialize the libraries used by the application. The details of initializing the main libraries are provided later in this document.

If there is no entry function, the application quits. Within the entry function, construct a main loop, and ensure that execution does not leave the entry function until the application quits.

## 5.3. Initializing the APPLET Library

The APPLET library is not just for using library applets. It is also required for handling HOME Button and POWER Button events and transitioning to Sleep Mode when the system is closed. Initialization of the APPLET library is performed before execution moves to the entry function. Initialization processing that must be performed by the application calls the `nn::applet::Enable()` function to enable each function of the APPLET library.

**Code 5-1. APPLET Library Initialization**

```
void nn::applet::Enable(bool isSleepEnabled = true);
```

Call this function after setting a sleep-related callback. Use *`isSleepEnabled`* when calling this function to specify whether a sleep-related callback is enabled.

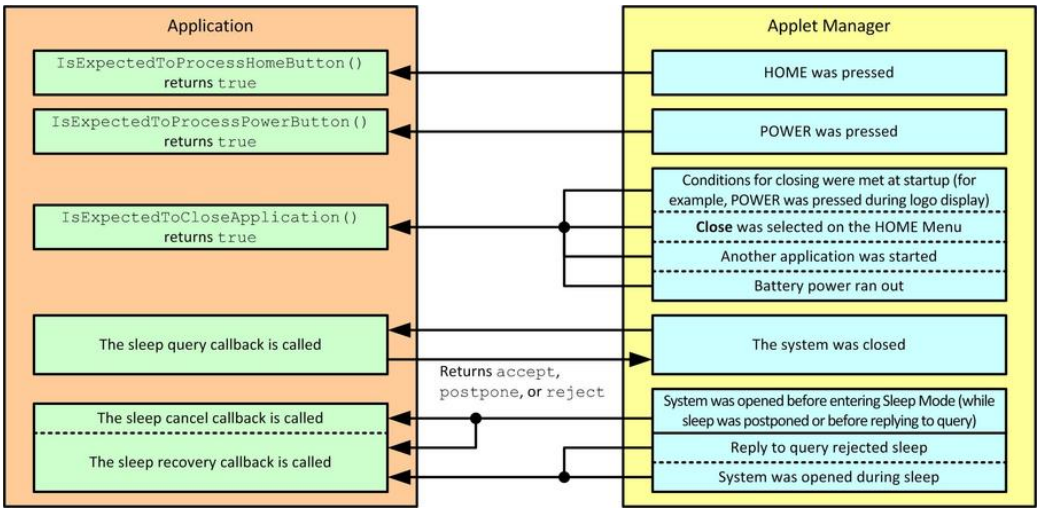
Just before and after calling this function, sleep-related handling must be carefully performed. For information about sleep-related handling, see 5.3.3. Sleep Handling.

> **Warning:** The `nn::applet::Enable()` function must be called before the `nngxInitialize`, `nn::dsp::Initialize`, or `nn::snd::Initialize()` functions.
>
> Primarily, this is because there are times when an application must be closed immediately after it starts due to something that occurred before the `Enable()` function is called, such as the POWER Button being pressed while the application was starting. For this reason, immediately after calling this function, handle with an application close request as described in 5.3.1. Handling Application Close Requests. At this time, do not call `nngxInitialize` until it is verified that the close request did not arrive.

The following sections explain application close requests, the HOME Button, sleep (closing system), and POWER Button handling that the application must implement. These sections also provide examples of handling these state changes. The system uses Applet Manager to notify the application about all state changes the application must respond to.

> **Warning:** When there has been no response to an application close request or a POWER Button press, the application is forcibly ended when the POWER Button is held down for longer than a certain period of time. The application must implement responses to these state changes so that they are handled by normal processing.

**Figure 5-1. Notifications Sent From the Applet Manager**


Application
IsExpectedToProcessHomeButton() returns true
IsExpectedToProcessPowerButton() returns true
IsExpectedToCloseApplication() returns true
The sleep query callback is called
The sleep cancel callback is called
The sleep recovery callback is called
Returns accept, postpone, or reject
Applet Manager
HOME was pressed
POWER was pressed
Conditions for closing were met at startup (for example, POWER was pressed during logo display)
Close was selected on the HOME Menu
Another application was started
Battery power ran out
The system was closed
System was opened before entering Sleep Mode (while sleep was postponed or before replying to query)
Reply to query rejected sleep
System was opened during sleep

## 5.3.1. Handling Application Close Requests

The Applet Manager uses the `nn::applet::IsExpectedCloseApplication()` function to send a notification if an application enters a state where it must close. This could occur, for example, when another application is started or **Close** is selected on the HOME Menu while an application is suspended. Applications must call this function periodically (such as once per frame). If `nn::applet::IsExpectedToCloseApplication` returns `true`, the application must quickly be closed. It is also possible for the application to enter a state where it has no rendering rights.

In this case, first make the application perform its own shutdown processing (within four seconds), and then call the `nn::applet::CloseApplication` to close the application. Because graphics-related processes cannot be executed from a state where there are no rendering rights, calls to `nngxInitialize` or `nngxWaitCmdlistDone` block processing. Also, command request completion interrupts are not generated in this state. You must implement the shutdown processing without waiting for completion of command requests.

The `nngxFinalize()` function can be called even in a state where the application does not have rendering rights, and calling it automatically releases the display buffer allocated by the `nngx` or `gl()` functions.

**Code 5-2. Functions Used to Close Applications**

```
bool nn::applet::IsExpectedToCloseApplication(void);
nn::Result nn::applet::CloseApplication(const u8* pParam=NULL,
                size_t paramSize=0, nn::Handle handle=NN_APPLET_HANDLE_NONE);
```

In addition to periodically calling the `IsExpectedToCloseApplication()` function, also make sure to call this function immediately after returning from the HOME Menu or a library applet to determine whether a condition requiring the application to close while waiting for the return has occurred. Also check immediately after initializing the APPLET library in case a close request has arrived while loading an application. If this function returns `true`, also execute the processes that you want to run when the application closes (for example, autosave).

## 5.3.1.1. Cautions When Shutting Down

Although the CTR-SDK is designed to release the allocated resources, even if an application calls `nn::applet::CloseApplication` at an arbitrary time, this feature has not been sufficiently tested. To safely close an application, implement closing based on the following measures until `CloseApplication` is called.

> **Note:** Revisions in future releases are expected to make the following measures unnecessary.

### Required

When an `nn::os::Alarm` object has been created, call the `Alarm` object member functions `Cancel()` and `Finalize()` in order.

When an `nn::os::Timer` object has been created, call the `Timer` object member functions `Stop()` and `Finalize()` in order.

Stop FS and UDS library processes, and then finalize the libraries. In particular, for the FS library, call the `Finalize()` function for each class being used. Even for NW4C and other non-SDK packages, be sure to finalize for classes being used by the FS library.

### Recommended

Finalize all libraries that have been initialized.

## 5.3.2. Handling HOME Button Presses

If the HOME Button is pressed, the Applet Manager uses the `nn::applet::IsExpectedToProcessHomeButton()` function to send notification of whether the application has entered a state that requires the HOME Menu to start. Applications must call this function periodically (such as once per frame). If this function returns `true`, the application must start the HOME Menu. Have the applications stop the operation of all devices and start the HOME Menu immediately. If, however, operations cannot be stopped due to a process that cannot be halted, you can display a HOME Disabled icon and stop the start of the HOME Menu. Note that the `IsExpectedToProcessHomeButton()` function continues to return `true` unless you use the `nn::applet::ClearHomeButtonState()` function to invalidate the fact that the HOME Button was pressed.

**Code 5-3. Functions Used to Start the HOME Menu**

```
bool nn::applet::IsExpectedToProcessHomeButton(void);
bool nn::applet::ProcessHomeButton(void);
nn::applet::AppletWakeupState nn::applet::WaitForStarting(
        nn::applet::AppletId* pSenderId=NULL, u8* pParam=NULL,
        size_t paramSize=0, s32* pReadLen=NULL, nn::Handle *pHandle=NULL,
        nn::fnd::TimeSpan timeout=NN_APPLET_WAIT_INFINITE);
void nn::applet::ClearHomeButtonState(void);
void nn::applet::ProcessHomeButtonAndWait();
```

Processing required to start the HOME Menu can be carried out merely by calling the `nn::applet::ProcessHomeButton()` function. If the return value is `true`, immediately call the `nn::applet::WaitForStarting()` function and wait for the return from the HOME Menu. Upon return, there are restrictions on the use of devices for purposes such as getting key input and rendering. Also, note that only the thread that called the `WaitForStarting()` function stops.

**Do not program an application in a way that would encourage operation that causes threads with priority level 16 or higher (a priority number from 1 to 16) to continue to occupy the CPU even after transitioning to the HOME Menu.If game memory is started up when a thread of this type is present, the system will freeze.**When an application thread continues to run while the HOME Menu is being displayed, performance for the HOME Menu and other operations deteriorate. We recommend that you program the system so that all application threads stop when transitioning to the HOME Menu.

The `nn::applet::ProcessHomeButtonAndWait()` function is a wrapper function that calls the `ProcessHomeButton()` function and handles waiting and Sleep Mode.

**Warning:**At some point before the `ProcessHomeButton()` function is called, you must call the `nngxWaitCmdlistDone()` function or perform equivalent processing to ensure that all GPU render commands have finished executing.

**Note:**The `ProcessHomeButton()` function can only be called during the time that the GX library can be used (from the completion of `nngxInitialize` until `nngxFinalize` is called).

Before you call the `ProcessHomeButton()` function, configure the display buffer, swap buffers, and call the `nngxStartLcdDisplay()` function to start LCD output. If LCD output has not been started, there is a chance that the HOME Menu could start up with black screens. Likewise, if the display buffer has not been configured or the buffers have not been swapped, there is a chance that undefined content could be displayed on the

screens.

**Table 5-1. Handling Devices During HOME Menu Display**

| Device | Application Response During HOME Menu Display |
|---|---|
| GPU/LCDs | Stop rendering, and do not update the display buffer. Also, by the time the HOME Menu is displayed, the graphics processing must be completed and the GPU must be in a stopped state. |
| Digital Buttons/Circle Pad | No particular handling is necessary. This input is not applied to the input values obtained by the application. |
| Touch Panel | No particular handling is necessary. This input is not applied to the sampling values obtained by the application; an invalid sampling value (0) is returned. |
| Accelerometer | No particular handling is necessary. Input values can be obtained. |
| Gyro Sensor | Input values can be obtained. If the gyro sensor will not be calibrated after returning from the HOME Menu, even during HOME Menu display, the application must get and correct these input values. |
| Sound | Functions in the SND library are designed to be called from the main thread and the sound thread. When calling functions from these threads, the application does not need to keep track of anything when transitioning with the HOME Button. However, if you are calling functions from other threads, you must provide appropriate handling so that calls are not made while the application is in a suspended state.<br>Because you cannot control the timing at which sound stops, the application must manage playback status or take other such steps to ensure synchronization of sound and graphics when such synchronization is necessary, such as when playing video. |
| Camera | No particular handling is necessary. Unless the cameras are finalized before HOME Menu startup, HOME Menu features that use the cameras will not work. |
| Microphone | No particular handling is necessary. |
| Wireless Communication | For non-local types of communication, no particular handling is necessary. However, if a browser or other application has been started and it uses wireless communication while the HOME Menu is displaying, that application's communication will be disconnected when the system returns from the HOME Menu.<br>Local communication can continue. However, you must consider the possibility that the system will be closed during HOME Menu display, which would result in a transition to Sleep Mode. If local communications are not ended at that point, the local communications status will transition to an error. Also, unless local communication is finalized before HOME Menu startup, any HOME Menu features that use local communication will not work. |
| NFC | When using the NFP library, call `nn::nfp::Finalize` to exit the NFP library before transitioning to the HOME Menu or an applet.<br>For more information, see the *3DS Programming Manual: NFP*. |

From immediately after returning from the HOME Menu until the application calls the `nngxSwapBuffers()` function to switch the display buffer, an image captured during transition to the HOME Menu is displayed on the screen. Note at this point is that although the content of main memory and VRAM is protected, GPU register settings must be reset by the application. Be sure to reset all register settings including the framebuffer, shader binary, and lookup tables, and not just the vertex load array settings and texture unit settings.

**Note:** The `nngxUpdateState(NN_GX_STATE_ALL)()` function call can be used to restore register settings as long as they are not directly overwritten.

### HOME Button Press Detection

Although detection of HOME Button presses can also be checked using notifications to a callback function registered using the `nn::applet::SetHomeButtonCallback()` function, implement this check by periodically checking the `nn::applet::IsExpectedToProcessHomeButton()` function.

**Code 5-4 Setting a HOME Button Callback Function**

```
void nn::applet::SetHomeButtonCallback(
        nn::applet::AppletHomeButtonCallback callback, uptr arg=0);
typedef bool (*nn::applet::AppletHomeButtonCallback)(
        uptr arg, bool isActive, nn::applet::CTR::HomeButtonState state);
```

Set a callback function by calling the `nn::applet::SetHomeButtonCallback()` function. Calls to the callback function pass the value of the *`arg`* parameter that was originally specified when `SetHomeButtonCallback` was called. The *`isActive`* parameter specifies whether an application is currently running, and the *`state`* parameter specifies the state of the HOME Button.

The status of the HOME Button is defined by the `nn::applet::HomeButtonState` enumerator type.

**Table 5-2. HOME Button States**

| Definition | Description |
|---|---|
| `HOME_BUTTON_NONE` | HOME Button is not pressed. |
| `HOME_BUTTON_SINGLE_PRESSED` | HOME Button is pressed (held for at least 200 ms). |

Use the `nn::applet::GetHomeButtonState()` function to check the state of the HOME Button. After a button press is detected, the HOME Button keeps that state until the next call to the `nn::applet::ClearHomeButtonState()` function. Calls to `ClearHomeButtonState` return the state of the HOME Button to `HOME_BUTTON_NONE`.

**Code 5-5. Checking the State of the HOME Button**

```
nn::applet::AppletHomeButtonState nn::applet::GetHomeButtonState(void);
```

The application can use the return value from the callback function to determine whether to enable or disable detection of HOME Button presses, and to control whether detection is indicated in the state of the HOME Button that is obtained by calling the `nn::applet::GetHomeButtonState()` function. Indicate detection of HOME Button presses for callback return values of `true`; do not for return values of `false`. Implement the callback function to return the value in the *`isActive`* parameter. Implement the function to only handle lightweight processing such as flag control; do not start the HOME Menu from directly within the callback function.

## 5.3.2.1. What an Application Can Do Before Starting the HOME Menu

After an application detects a HOME Button press, it has a maximum of 0.5 seconds until it must start the HOME Menu. Within this time, the application can carry out such operations as pausing the action or auto-saving. When you do not want to have a static image of the game as the HOME Menu background (such as for timed puzzle games), you could use this time to process a different image to hide the game screen. But in general, we recommend using just a static game screen as the background.

When an application is suspended, the rendering process to create a capture image to display on the upper screen of the HOME Menu is performed after it is confirmed that `nn::applet::IsExpectedToProcessHomeButton` returns `true`. However to display the HOME Menu with `nn::applet::ProcessHomeButton`, the GPU processing must be stopped (graphics processing must be completed). Call `nngxWaitCmdlistDone` to wait until all graphics commands have completed. After that, call `nngxWaitVSync` to ensure that the images displayed on the LCDs are updated.

When starting the HOME Menu, consider the possibility that the user will close the application in the HOME Menu and implement it so that there are no problems even if the application is closed in the HOME Menu. Although you can usually perform shutdown processing after control returns to the application, note that shutdown processing becomes impossible if the battery runs down during Sleep Mode while the HOME Menu is being displayed.

## 5.3.2.2. Displaying the HOME Menu Disabled Icon

When an application detects a HOME Button press while in the middle of a process that cannot be interrupted (such as saving), and the HOME Menu cannot be displayed right away, the application can display the HOME Menu Disabled icon in the middle of the lower screen and cancel starting the HOME Menu.
Implement the HOME Menu Disabled icon according to the following specifications.

- Icon: Image file included in the CTR-SDK (`HomeNixSign_Targa.tga`)
- Display position: Center of the lower screen
- Fade-in: 0.083 seconds (5 frames at 60 fps)
- Display time: 1 second (60 frames at 60 fps)
- Fade-out: 0.333 seconds (20 frames at 60 fps)

If the user presses the HOME Button while the icon is displayed, simply continue displaying the icon. For example, if the user presses the HOME Button while the icon is fading out, you do not need to fade in again. Just continue with the fade-out. Even if it does become possible to start the HOME Menu while the HOME Menu Disabled icon is still displaying, you must not start the HOME Menu while the icon is displayed.

## 5.3.2.3. Prohibiting the Posting of Screenshots

You can prevent other applications (such as system applets like Miiverse) from posting images created when transitioning to the HOME Menu.

**Code 5-6. Setting and Getting Screen Capture Post Permissions**

```
nn::Result nn::applet::SetScreenCapturePostPermission(
    const nn::applet::ScreenCapturePostPermission permission);
nn::Result nn::applet::GetScreenCapturePostPermission(
    nn::applet::ScreenCapturePostPermission* pPermission);
```

You can specify whether posting is allowed with the `SetScreenCapturePostPermission()` function. Two values can be specified for the *`permission`* argument: `SCREEN_CAPTURE_POST_ENABLE` or `SCREEN_CAPTURE_POST_DISABLE`. Specifying any other

values is prohibited. If the application is restarted with the `nn::applet::RestartApplication()` function, the initial values (see Table 5-3) are restored.

You can get the current settings with the `GetScreenCapturePosetPermission()` function.

**Table 5-3. Values Used When Setting and Getting Screen Capture Post Permissions**

| Definition | Description |
|---|---|
| `SCREEN_CAPTURE_POST_NO_SETTING` | (Specifying this value is prohibited.) The default value, where posting is allowed. |
| `SCREEN_CAPTURE_POST_NO_SETTING_OLD_SDK` | (Specifying this value is prohibited.) The default value obtained by applications released under an old version of the SDK, where posting is allowed. Posting screenshots is prohibited when the camera is in use. The camera is in use after `nn::camera::Initialize` is called to initialize the camera, until `nn::camera::Finalize` is called to close the camera from the CAMERA library. |
| `SCREEN_CAPTURE_POST_ENABLE` | Posting is allowed. |
| `SCREEN_CAPTURE_POST_DISABLE` | Posting is prohibited. |

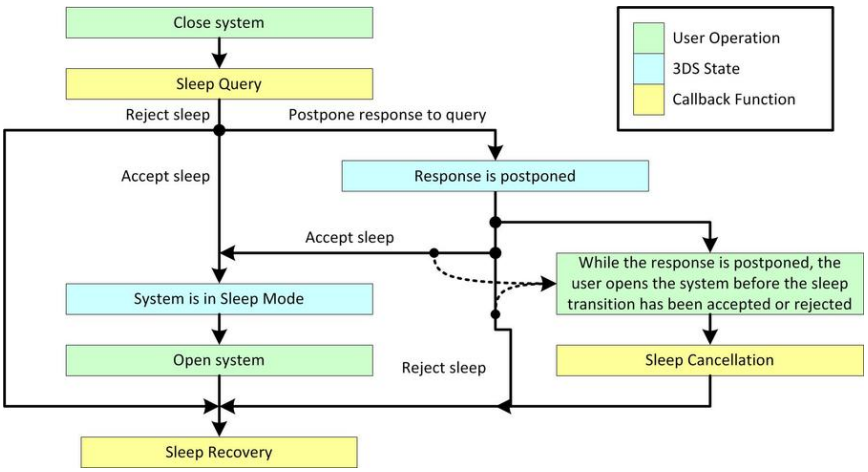
**Note:** Contact Nintendo support when you want to disable screenshot posting to Miiverse from an application using a CTR-SDK version earlier than 7.x.

## 5.3.3. Sleep Handling

As with the DS, on 3DS an application can determine whether to transition to Sleep Mode when the system is closed while the application is running. However, with 3DS it is possible to transition to Sleep Mode even if the application is suspended, such as when the HOME Menu is being displayed.

The following figure indicates sleep state transitions as the relationship between the related user operations, 3DS state, and callback function association. This section describes how to handle sleep transitions focused on these callback functions and then provides information associated with sleep.

**Figure 5-2 Sleep State Transitions**



### 5.3.3.1. Sleep Query Callback

When the system is closed while an application is running, the Applet Manager queries the application about whether to transition to Sleep Mode by system closing. The application sends a notification in reply to this query using the set callback function.

**Code 5-7. Setting a Sleep Query Callback Function**

```
void nn::applet::SetSleepQueryCallback(
                 nn::applet::AppletSleepQueryCallback callback, uptr arg=0);
typedef nn::applet::AppletQueryReply
                 (*nn::applet::AppletSleepQueryCallback)(uptr arg);
```

Set a callback function by calling the `nn::applet::SetSleepQueryCallback()` function. Calls to the callback function pass the value for the *arg* parameter that was originally specified to `SetSleepQueryCallback`.

The application notifies the Applet Manager whether to transition immediately to Sleep Mode depending on the query response.

**Table 5-4. Values Specifiable as the Return Value of the Sleep Query Callback Function**

| Definition | Description |
|---|---|
| REPLY_REJECT | Rejects the transition to sleep. |
| REPLY_ACCEPT | Accepts transition to sleep. |
| REPLY_LATER | Postpones the transition to sleep. |

Return REPLY_REJECT when the application continues with wireless communication or sound playback after the system is closed. Return REPLY_ACCEPT to transition directly to Sleep Mode, but note that the system does so immediately and the transition may not necessarily occur at a time that is safe for graphics and other processing that is underway. Return REPLY_LATER to postpone transitioning to Sleep Mode until the application can safely pause other processing. However, when doing so, the application must call `nn::applet::ReplySleepQuery()` and pass REPLY_ACCEPT as an argument as soon as it is safe to transition to sleep, and then use an event class or another mechanism to wait for the system to wake up again.

When REPLY_LATER is returned, the system is in a state where it is ready to transition to Sleep Mode, such as turning off LCD power. Also, because the Applet Manager is in a semi-stopped state where it cannot perform HOME Button or POWER Button processing, use as little processing time as possible when stopping other processes. However, there is no problem with continuing the postponed state for several tens of seconds as long as the sleep cancel callback (see 5.3.3.3. Sleep Cancel Callback) is used to appropriately handle cases where the system is opened while sleep is postponed. If you do not use a sleep cancel callback to handle such cases, system behavior is not considered a violation of the CTR Guidelines as long as sleep is postponed for no longer than four seconds. If you want to prohibit sleep for longer than four seconds, use a combination of REPLY_REJECT and the `nn::applet::EnableSleep()` function (described below), and reissue the sleep query callback.

**Warning:** Do not accept transitions to Sleep Mode while initializing local communications (while executing the `nn::uds::Initialize()` function). Conflicts with wireless communications state transitions could result in the possibility of Sleep Mode transition not occurring correctly. Also, when you transition to Sleep Mode without

finalizing local communications, the UDS library forcibly disconnects local communications and transitions to an error state. Once in this error state, UDS library functions, with some exceptions, will return errors until finalization takes place.

When calling functions in the SND library from threads other than the main or sound threads, you must apply appropriate controls so that no SND library functions are called between when the transition to sleep is accepted with REPLY_ACCEPT and when the system recovers from the sleep state.

In some cases, sound thread processes get jammed up during the transition to the sleep state. In these cases, if the time between calls to the `nn::snd::WaitForDspSync()` and `nn::snd::SendParameterToDsp()` functions (which are normally made every five milliseconds) exceeds 100 milliseconds, sometimes the sound thread remains stopped when the system recovers from the sleep state. There have been infrequent reports of threads stopping in this way in Debug builds only.

If the battery runs out during sleep, the system never recovers from sleep and finalization is impossible. Also, in some cases the game card will be removed during sleep. You must keep both of these cases in mind and implement your code so that there is no problem even if the application terminates on an error during sleep.

If a sleep query callback function has not been registered by the application, status is the same as when a callback function that returns REPLY_REJECT has been registered.

### Callbacks While an Application Is Suspended

The sleep query callback function can be called even if the application is not running, such as when the HOME Menu is being displayed. The `nn::applet::IsActive()` function can be used to check whether an application is running. Return REPLY_ACCEPT when it is determined in the sleep query callback function that the application is suspended.

**Code 5-8. Determining Whether an Application Is Running**

```
bool nn::applet::IsActive(void);
```

If the return value is `true`, the application is running. If the value is `false`, it is suspended.

The state in which application operations are suspended is called the Inactive state (as opposed to the Active state when the application is running). As when the HOME Menu or library applet is being displayed, threads other than those used by the applications to call functions to display them or to wait for them to complete can continue to run. Also, during the interval until `nn::applet::Enable` is called while the application is running, no sleep-related callbacks are issued, but the application is in the Inactive state.

During the Inactive state, the VSync callback and other callbacks including the sleep query callback can be issued as usual. However, because it is usual to implement so that the main thread stops in the Inactive state, in some cases an unintentional deadlock state occurs because a sleep query callback cannot be handled properly. For example, when sleep handling is performed by the main thread, if REPLY_LATER is returned while in the Inactive state, there is no opportunity to accept the transition to the sleep state because the main thread is stopped and the application state stagnates.

Excluding the case when REPLY_REJECT is always returned during communications, we recommend stopping access to the FS library before entering the Inactive state, such as when

displaying the HOME Menu, and always returning REPLY_ACCEPT to sleep queries while in the Inactive state. When creating threads to run even during the Inactive state, there is no problem in using the same processing as that in the Active state as long as there is appropriate sleep handling.

### Controlling Sleep Responses

You can control whether the application supports transition to Sleep Mode using the `nn::applet::EnableSleep` and `nn::applet::DisableSleep()` functions.

**Code 5-9. Controlling Sleep Mode Support**

```
void nn::applet::EnableSleep(
                        bool isSleepCheck=nn::applet::SLEEP_IF_SHELL_CLOSED);
void nn::applet::DisableSleep(
                        bool isReplyReject=nn::applet::REPLY_REJECT_IF_LATER);
```

The `nn::applet::EnableSleep()` function validates the return value specified by the sleep query callback function, allowing control to move to the Sleep Mode transition sequence. If `SLEEP_IF_SHELL_CLOSED` is specified in *`isSleepCheck`*, the system status is checked when the function is called. The sleep query callback function is called if the system is closed. If `NO_SHELL_CHECK` is specified in *`isSleepCheck`*, there is no difference in operations except that the system status is not checked and no callback function is called. Incidentally, when sleep-related callbacks are enabled with `nn::applet::Enable(true)`, `EnableSleep(NO_SHELL_CHECK)` is executed within the callback functions.

If `SLEEP_IF_SHELL_CLOSED` is specified in *`isSleepCheck`*, and `nn::applet::EnableSleep()` is called, it is not necessary to consider the case of the system being opened while transitioning to the sleep state because a sleep query callback is issued while the system is closed. If `REPLY_REJECT` is returned rather than `REPLY_LATER` in the sleep query callback functions, necessary processing is performed while sleep is being rejected. In other words, if `REPLY_REJECT` is returned using the sleep query callback, there are no sleep query callbacks issued while the system is closed, but by calling `EnableSleep(SLEEP_IF_SHELL_CLOSED)` when the current state is ready to transition to sleep state, the sleep query callback can be issued again while the system is still closed. Even if `DisableSleep()` is not called, `EnableSleep()` can be called at any time. However, several milliseconds may pass from when a function is called until it returns depending on the Applet Manager state, so we do not recommend calling in every frame.

> **Warning:** When the system is closed while an application is running, it is the equivalent of returning `REPLY_REJECT` to the sleep query until `nn::applet::Enable` is called. For applications that support sleep, call `EnableSleep(SLEEP_IF_SHELL_CLOSED)` after `Enable` to be able to reissue sleep query callbacks even if the system is closed during operations.

The `nn::applet::DisableSleep()` function disables the return value specified by the sleep query callback function, setting the status to `REPLY_REJECT` regardless of which return value is specified. If `REPLY_REJECT_IF_LATER` is specified in *`isReplyReject`*, the `nn::applet::ReplySleepQuery(REPLY_REJECT)()` function is executed and the request is rejected if the system has not already attempted to enter Sleep Mode. If `NO_REPLY_REJECT` is specified in *`isReplyReject`*, there is no difference in operations except that the `ReplySleepQuery(REPLY_REJECT)()` function is not executed.

A sleep query callback can be issued when the system is closed even after `DisableSleep` has been called, but the callback function return value is ignored and it is considered as if REPLY_REJECT is always returned. However, note that the return values of sleep query callbacks called while the application is suspended (state when `nn::applet::IsActive` returns `false`) are valid (can return values other than REPLY_REJECT). Also, `DisableSleep` does not have an internal counter, so even if it is called multiple times, calling `EnableSleep` once makes the callback function's return value valid.

When calling a function that suspends the application, such as displaying the HOME Menu (`nn::applet::ProcessHomeButton`) or POWER Menu (`nn::applet::ProcessPowerButton`) or starting a library applet, before you call this function be sure to first reject the sleep query with `DisableSleep`, and then call `EnableSleep` after the application recovers.

> **Warning:** If the `DisableSleep(REPLY_REJECT_IF_LATER)()` function is executed before performing application shutdown processing, execution may stop during Sleep Mode if the system is closed during shutdown processing.

#### Checking the Notification State and Replying to Postponed Queries

To determine whether transition to sleep has been postponed in the application, call `nn::applet::IsExpectedToReplySleepQuery`. If this function returns `true`, the application must determine whether it is ready to transition to the sleep state and must use the `nn::applet::ReplySleepQuery()` function to reply to the postponed query.

In your implementation of handling transitions to sleep state, we recommend that you call the `IsExpectedToReplySleepQuery()` function periodically (such as once per frame) and then transition to sleep state at some predetermined time.

**Code 5-10. Checking Sleep Notification State and Replying to Postponed Queries**

```
bool nn::applet::IsExpectedToReplySleepQuery(void);
void nn::applet::ReplySleepQuery(nn::applet::AppletQueryReply reply);
```

## 5.3.3.2. Sleep Recovery Callback

The application uses a callback function to receive notification that the system has opened and the system has recovered from sleep.

**Code 5-11. Setting a Sleep Recovery Callback Function**

```
void nn::applet::SetAwakeCallback(nn::applet::AppletAwakeCallback callback,
                                  uptr arg=0);
typedef void (*nn::applet::AppletAwakeCallback)(uptr arg);
```

Call `nn::applet::SetAwakeCallback` to set a sleep recovery callback function. Calls to the callback function pass the value for the *arg* parameter that was originally specified to `SetAwakeCallback`.

Have this callback function only perform simple processing, such as signaling the event class instance that is waiting to resume. This callback function is called immediately if sleep status

does not result even though the system is closed for some reason such as REPLY_REJECT being returned by the sleep query callback function. Whether the system has been opened cannot be determined by calling this callback function.

**Warning:** The LCD displays are turned off when the system transitions to Sleep Mode. Consequently, when the system is ready to display a screen after waking up, the screens do not show anything until the `nn::gx::StartLcdDisplay()` function is called. Note, however, that the screen display may momentarily distort if the `nn::gx::StartLcdDisplay()` function is called while recovering from sleep while library applets are executing. If this happens, make the call after preparations for screen display have been completed and a state allowing normal display has been established.

## 5.3.3.3. Sleep Cancel Callback

You can use the `nn::applet::SetSleepCanceledCallback()` function to register a callback function to be called in case sleep is canceled. Sleep cancellation occurs when the system has been closed and is opened again before the application can transition to Sleep Mode.

**Code 5-12. Registering a Sleep Cancel Callback Function**

```
void nn::applet::SetSleepCanceledCallback(
                nn::applet::AppletSleepCanceledCallback callback, uptr arg=0);
typedef void (*nn::applet::AppletSleepCanceledCallback)(uptr arg);
```

Use this callback function in cases such as when halting the sleep process by detecting if the system has been opened while performing processing that takes time. This can happen in cases such as saving data before entering Sleep Mode after the system is closed.

If nothing is done inside this callback function, the transition to Sleep Mode is postponed. In other words, Sleep Mode will not be canceled unless the `ReplySleepQuery(REPLY_REJECT)()` function is called explicitly. Also, note that the sleep recovery callback function will be called even if this callback function is called.

This callback function does not need to be registered when transitioning to Sleep Mode immediately or when the time required for other processing to complete is short. In these cases, you may consider not implementing the sleep cancel callback feature.

Normally, a sleep cancel callback is issued only when REPLY_LATER is returned. However, when the system is quickly opened and closed, a sleep cancel callback may be issued after REPLY_ACCEPT or REPLY_REJECT is returned. Reproducing when the system is opened before a reply is sent to the sleep query callback may be difficult to do during debug because this error can only be reproduced when the cover is opened and closed quickly.

Sleep-related callbacks are not called simultaneously or in parallel. It is guaranteed that the sleep cancel callback will be issued either zero times or one time between the sleep query callback and the sleep recovery callback.

**Note:** The reference for the `nn::applet::SetSleepCanceledCallback()` function shows a sample implementation.

## 5.3.3.4. Prohibited Processes While in Sleep Mode

**Note:** Currently, no processes are prohibited to applications while the system is in Sleep Mode.

## 5.3.3.5. Device State When the System Is Closed

Some devices work differently when the system is closed depending on whether the argument passed when calling `nn::applet::ReplySleepQuery` is `REPLY_ACCEPT` or `REPLY_REJECT`.

**Table 5-5. Device State When the System Is Closed**

| Device | REPLY_ACCEPT | REPLY_REJECT |
|---|---|---|
| LCD | Display buffer updates and VSyncs are both stopped, and LCD displays are turned off. | Display buffer updates are stopped and LCD backlights are turned off (but screen displays are maintained). |
| Digital buttons | No input accepted. | CTR only accepts input from the L and R Buttons. SNAKE accepts input from the L, R, ZL, and ZR Buttons. |
| Touch Panel | Sampling values cannot be obtained. | Returns an invalid sampling value (0). |
| Accelerometer | Functions as a pedometer. Input values cannot be obtained. | Functions as a pedometer. Input values can be obtained. |
| Gyro Sensor | Suspended. | Input values can be obtained. |
| Sound | Suspended. | Normally output is suspended from the speakers and forcibly output from the headphones, but you can make settings so that the output also comes from the speakers. |
| Camera | Suspended. | Suspended. |
| Microphone | Suspended. | Suspended. |
| Wireless Communication | Suspended. Recommend suspending local communication ahead of time. | Not suspended. |
| Infrared Communication | Suspended. We recommend running the disconnect process and waiting for it to complete before entering Sleep Mode. | Not suspended. |
| NFC | Suspended. | Suspended. |

Threads created by an application, including the main thread, are all suspended when the system transitions to Sleep Mode. If the system has not transitioned to Sleep Mode, the threads continue running, but will no longer receive events or other notifications from suspended devices.

**Warning:** Unlike with the DS/DSi systems, note that sound is not normally output from the speakers when the system is closed. For information about having output come from the speakers, see 10.5.7. Sound Output When the System Is Closed and Sleep Is Rejected.

## 5.3.4. Handling the POWER Button

When the POWER Button is pressed for 160 ms or longer, the Applet Manager uses the `nn::applet::IsExpectedToProcessPowerButton()` function to send notification whether the application needs to handle the fact that the POWER Button was pressed. Applications must call this function periodically (such as once per frame). If this function returns `true`, call the `nn::applet::ProcessPowerButton()` function as soon as feasible.

**Code 5-13. Functions Used to Handle POWER Button Presses**

```
bool nn::applet::IsExpectedToProcessPowerButton(void);
bool nn::applet::ProcessPowerButton(void);
void nn::applet::ProcessPowerButtonAndWait();
```

After you call the `nn::applet::ProcessPowerButton()` function, control transfers to the system in order to display the POWER Menu. By displaying the POWER Menu, there is no need for the application to display a special screen when closing. Immediately after calling this function, the application must begin waiting with the `nn::applet::WaitForStarting()` function for permission to begin finalization. After processing returns from `WaitForStarting`, because `nn::applet::IsCloseApplication` returns `true` in this state, perform the application close process while referring to 5.3.1. Handling Application Close Requests.

The `nn::applet::ProcessPowerButtonAndWait()` function is a wrapper function that calls the `ProcessPowerButton()` function and handles waiting and Sleep Mode.

**Note:** The `ProcessPowerButton()` function can only be called while the GX library can be used (from when `nngxInitialize` completes until `nngxFinalize` is called).

Before you call the `ProcessPowerButton()` function, configure the display buffer, swap buffers, and call the `nngxStartLcdDisplay()` function to start LCD output. If LCD output has not been started, there is a chance that the Power Menu could start up with black screens. Likewise, if the display buffer has not been configured or the buffers have not been swapped, there is a chance that undefined content could be displayed on the screens.

## 5.3.5. Example of Handling by the Application

Implement sleep or HOME Button handling based on the following points.

- Perform the following determinations in locations where the main loop is called periodically.
  - Evaluate the `nn::applet::IsExpectedToProcessHomeButton()` function:
    When `true` is returned, call `nn::applet::ProcessHomeButton`.
    (When you do not want to or cannot display the HOME Menu, display the HOME Menu Disabled icon according to the guidelines.)

  - Evaluate the `nn::applet::IsExpectedToProcessPowerButton()` function:
    When `true` is returned, call `nn::applet::ProcessPowerButton`.

  - Evaluate the `applet::IsExpectedToCloseApplication()` function:

If `true`, close the application.

- Before calling the `nn::applet::ProcessHomeButton` or `nn::applet::ProcessPowerButton()` function, you must ensure that all render commands have finished executing. Use the `nngxWaitCmdlistDone()` function to wait for currently executing render commands to finish.
- After you call the `nn::applet::ProcessHomeButton` or `nn::applet::ProcessPowerButton()` function, you must always call the `nn::applet::WaitForStarting()` function.
- When control is returned from `nn::applet::Enable` or `nn::applet::WaitForStarting`, be sure to perform `nn::applet::IsExpectedToCloseApplication` determination.

The following sample code shows how to handle sleeping and the HOME Button.

**Code 5-14. Sample Code for Handling Different Situations**

```
nn::os::LightEvent sAwakeEvent(true);
nn::os::LightEvent sTransitionEvent(true);
nn::os::CriticalSection sFileSystemCS(WithInitialize);

// Sleep Query callback
nn::applet::AppletQueryReply mySleepQueryCallback(uptr arg)
{
    sAwakeEvent.ClearSignal();
    return (nn::applet::IsActive() ? REPLY_LATER : REPLY_ACCEPT);
}

// Sleep wake up callback
void myAwakeCallback(uptr arg)
{
    sAwakeEvent.Signal();
}

// Sleep
void sleepApplication()
{
    // Lock to prevent Access from the FS library during sleep
    // If lock fails, retry in next frame
    if (sFileSystemCS.TryEnter())
    {
        _app_prepareToSleep();
        nn::applet::ReplySleepQuery(REPLY_ACCEPT);
        sAwakeEvent.Wait();
        _app_recoverFormSleep();
        sFileSystemCS.Leave();
        nn::gx::StartLcdDisplay();
    }
}

// Application finalization.
void exitApplication()
{
    _app_finalize();
    nn::applet::CloseApplication();
}

// Main loop.
void nnMain()
{
    // Run normally when an event is in the signaled state.
    sAwakeEvent.Signal();
    sTransitionEvent.Signal();
    // Set callbacks.
    nn::applet::SetSleepQueryCallback(mySleepQueryCallback);
    nn::applet::SetAwakeCallback(myAwakeCallback);
    nn::applet::Enable(true);
    // Handle close requests while the application is loading.
    if (nn::applet::IsExpectedToCloseApplication())
    {
        exitApplication();
    }
```

```
    // Handle the system being closed while the application is loading.
    nn::applet::EnableSleep(SLEEP_IF_SHELL_CLOSED);

    while (true)
    {
        _app_exec();

        // Reply to Sleep Query.
        if (nn::applet::IsExpectedToReplySleepQuery())
        {
            if (_app_isRejectSleep())
            {
                // Notify of rejection if unable to enter Sleep Mode.
                nn::applet::ReplySleepQuery(REPLY_REJECT);
            } else {
                sleepApplication();
            }
        }
        // Application Close Requests
        if (nn::applet::IsExpectedToCloseApplication())
        {
            exitApplication();
        }
        // HOME Menu
        if (nn::applet::IsExpectedToProcessHomeButton())
        {
            if (_app_isSuppressedHomeButton())
            {
                _app_drawSuppressedHomeButtonIcon();
                nn::applet::ClearHomeButtonState();
            } else {
                // During state transitions that stop the main thread,
                // such as starting the HOME Menu or a library applet,
                // set the event to an unsignaled state.
                sTransitionEvent.ClearSignal();
                // When the system enters Sleep Mode while the HOME Menu is
displayed,
                // lock to prevent access by the FS library.
                // If unable to acquire lock, try again next frame.
                if (sFileSystemCS.TryEnter())
                {
                    _app_prepareToHomeButton();
                    nn::applet::ProcessHomeButtonAndWait();
                    sFileSystemCS.Leave();
                    if (nn::applet::IsExpectedToCloseApplication())
                    {
                        exitApplication();
                    }
                    sTransitionEvent.Signal();
                    // The GPU register settings must be restored.
                    _app_recoverGpuState();
                }
            }
        }
        // POWER Button
        if (nn::applet::IsExpectedToProcessPowerButton())
        {
            // Lock to prevent access by the FS library.
            // If unable to acquire lock, try again next frame.
            if (sFileSystemCS.TryEnter())
            {
                nn::applet::ProcessPowerButtonAndWait();
                sFileSystemCS.Leave();
                if (nn::applet::IsExpectedToCloseApplication())
                {
                    exitApplication();
                }
                // The GPU register settings must be restored.
                _app_recoverGpuState();
            }
        }
    }
}
// Lock as demonstrated below before accessing via the FS library
// from outside the main thread.
{
    // If in Sleep Mode, stop the thread until the system wakes.
    sAwakeEvent.Wait();
    // Stop the thread until control returns to the main thread.
    s_TransitionEvent.Wait();
    {
```

```
            os::CriticalSection::ScopedLock lock(sFileSystemCS);
            //
            // Write the process to access FS here.
            //
        }
    }
```

## 5.3.6. Restarting an Application

You can call the `nn::applet::RestartApplication()` function to restart an application without returning to the HOME Menu. You can specify the parameters to pass to the application after it is restarted, and you can use the `nn::applet::GetStartupArgument()` function to get these parameters.

**Code 5-15. Functions Used to Restart an Application**

```
nn::Result nn::applet::RestartApplication(
                const void* pParam = NULL,
                size_t paramSize = NN_APPLET_PARAM_BUF_SIZE);
bool nn::applet::GetStartupArgument(
                void* pParam,
                size_t paramSize = NN_APPLET_PARAM_BUF_SIZE);
```

Specify a byte array of parameters in *pParam* and the size, in bytes, of the parameters in *paramSize*. The array cannot be larger than NN_APPLET_PARAM_BUF_SIZE.

The `nn::applet::RestartApplication()` function usually restarts an application without returning. To find out whether this function was used to restart an application, check the return value from the `nn::applet::GetStartupArgument()` function, which returns `true` when it can get the parameters that were passed to the restarted application.

## 5.3.7. Jump to System Settings

This function allows a jump directly from an application to System Settings screens for Internet Settings, Parental Controls, or Data Management.

**Code 5-16. Jump to System Settings**

```
nn::Result nn::applet::JumpToInternetSetting(void);
nn::Result nn::applet::JumpToParentalControls(
                nn::applet::AppletParentalControlsScene scene =
                nn::applet::CTR::PARENTAL_CONTROLS_TOP);
nn::Result nn::applet::JumpToDataManagement(
                nn::applet::AppletDataManagementScene scene =
                nn::applet::CTR::DATA_MANAGEMENT_STREETPASS);
bool nn::applet::IsFromMset(nn::applet::AppletMsetScene* pScene = NULL);
```

An application can jump to the Internet Settings, Parental Control, or Data Management settings screens by calling the `nn::applet::JumpToInternetSetting`, `nn::applet::JumpToParentalControls`, or `nn::applet::JumpToDataManagement()` function, respectively. The application shuts down before jumping to System Settings, so perform shutdown processing in advance. Any failure in calls to these functions results in a fatal error. Also, regardless of whether these functions are successful, control does not return to the application. The Parental Control screen contains multiple configuration items.

A parameter is provided to allow selection of the jump destination.

**Table 5-6. Specifying Jump Destination Screen**

| Definition | Jump Destination Screen |
|---|---|
| PARENTAL_CONTROLS_TOP | Parental control main menu |
| PARENTAL_CONTROLS_COPPACS | Parental control COPPACS authentication processing screen |
| DATA_MANAGEMENT_STREETPASS | StreetPass setting screen for data management screen |

When the system setting application is exited after one of these jump functions, the application is restarted. During a restart after exiting system settings, a call to the `nn::applet::IsFromMset()` function returns `true`. To determine which scene was jumped to in the system settings, the *pScene* parameter specifies the variable that stores the jump target. Call the `nn::applet::IsFromMset()` function after the `nn::applet::Enable()` function is called.

## 5.3.8. Initial Parameters That Can Be Obtained at Startup

In cases where restarts happen from applications or from System Settings after a jump, the following functions determine how the application was restarted.

**Table 5-7. Functions That Get the Initial Parameters When Restarting**

| Function | Initial Parameters |
|---|---|
| `nn::applet::GetStartupArgument` | Parameters specified with the `nn::applet::RestartApplication()` function. |
| `nn::applet::IsFromMset` | The System Settings screen jumped to from the application. |
| `nn::news::IsFromNewsList` | The type of notification and parameters set. The user is notified if **Start Software** is selected from the notifications list. |
| `nn::friends::IsFromFriendList` | Friend's friend key. The user is notified if **Join Game** is selected. |

## 5.3.9. Jump to Nintendo eShop

You can also make jumps from applications to pages in Nintendo eShop.

**Code 5-27. Verifying the Nintendo eShop Installation**

```
bool nn::applet::IsEShopAvailable();
```

Depending on which system updates have been applied, Nintendo eShop might not be installed on the user's 3DS system. Be sure to call the `nn::applet::IsEshopAvailable()` function before any jump to make sure that Nintendo eShop has been installed.

**Note:** If Nintendo eShop is not on the user's system, notify the user that the system needs to be updated on the Internet.

For example, display the message, "Nintendo eShop is unavailable. Please update your system on the Internet to use Nintendo eShop." ."

**Note:** The install status does not need to be checked for downloadable applications.

**Code 5-18. Jump to Nintendo eShop (Details Page)**

```
nn::Result nn::applet::JumpToEShopTitlePage(bit32 uniqueId);
```

The page shown after the jump is the details page for the title with the unique ID specified by the *uniqueId* parameter. If the title specified is not one that can be publicly searched (because its name is not shown in searches), the page is not shown. Also, if the *uniqueId* parameter is set to a unique ID for a title that has not been registered in Nintendo eShop or an invalid value, an error is shown in Nintendo eShop.

**Code 5-19. Jump to Nintendo eShop (Patch Page)**

```
nn::Result nn::applet::JumpToEShopPatchPage(bit32 uniqueId);
```

When an error that an application must be updated, such as `nn::act::ResultApplicationUpdateRequired`, is returned from the authentication server or account server, use this to navigate to the patch page for the Nintendo eShop title.

The page shown after the jump is the patch page for the title with the unique ID specified by the *uniqueId* parameter.

There is no patch page in Nintendo eShop for a title when application updates are handled by remaster. In this case, a "Title does not exist" error is displayed and the Nintendo eShop startup screen appears. Use `nn::applet::JumpToEShopTitlePage()` to handle this operation by remaster.

Because these functions that jump to Nintendo eShop end the application when they are called, run exit processing before calling them. The application is not restarted after Nintendo eShop is shut down.

**Note:** When using this function to jump, submit the jump target title to OMAS. For more information, see the *Guidelines*: *e-Commerce*.

## 5.3.10. Jump to E-manual

It is possible to jump from an application to the e-manual that users start from the HOME Menu.

**Code 5-20. Jump to E-manual**

```
void nn::applet::JumpToManual();
```

After the jump, the screen shows the application's manual.

The function jumps after suspending the application, so after calling the function, use the

`nn::applet::WaitForStarting()` function to wait for resumption from the HOME Menu. The HOME Menu resumes the application after the e-manual closes.

## 5.4. Initializing the FS Library

In many cases, the next library to initialize is the FS library, which allows access to files on media. Call the `nn::fs::Initialize()` function to initialize the FS library. You must use the classes provided by the FS library to access files on media.

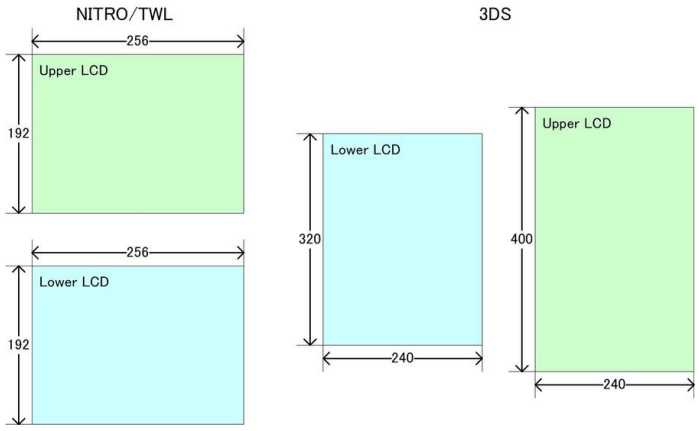
## 5.5. Initializing the GX Library

Use the GX library to draw to the LCD screen or render 3D graphics. Call the `nngxInitialize()` function to initialize the GX library. During initialization, you must specify functions for handling memory allocation and release requests from the library.

After calling the `nngxInitialize()` function, do any other required operations, such as creating command-list objects for executing graphics commands or allocating memory for the display and render buffers for displaying on the LCD.

Note that the 3DS LCD layouts and resolutions differ from those of the NTR or TWL, as shown in Figure 5-3.

**Figure 5-3. LCD Layouts and Display Resolutions**




This figure shows an implementation example in a sample program. For more information about the features and settings used, see the separate *3DS Programming Manual: Basic Graphics* and the *CTR-SDK API Reference*. For more information about stereoscopic display, see the *3DS Programming Manual: Advanced Graphics*.

### 5.5.1. Initializing the Library

When calling the `nngxInitialize()` function, you must specify the memory allocator and

deallocator functions that will handle memory requests from the library.

**Code 5-21. Initializing the GX Library**

```
void SetupNngxLibrary(const uptr fcramAddress, const size_t memorySize)
{
    InitializeMemoryManager(fcramAddress, memorySize);

    if (nngxInitialize(GetAllocator, GetDeallocator) == GL_FALSE)
    {
        NN_PANIC("nngxInitialize() failed.\n");
    }
}
```

## 5.5.1.1. Allocator

The memory accessed by the GPU for the texture image data and rendering buffer needed for graphics processing is allocated using the allocator function specified in the call to the `nngxInitialize()` function.

The allocator takes four arguments.

### First Argument (Allocation Memory Space)

The first argument specifies the memory space for allocation, passed as a `GLenum` type. The memory space for allocation depends on the value passed.

When passing `NN_GX_MEM_FCRAM`, the memory region is allocated from main memory, and in such cases, this memory must be allocated from the device memory portion of main memory. Device memory is a region in main memory for which the operating system guarantees address integrity when it is accessed by both peripheral devices and the main process. For more information, see 3.1.2. Device Memory.

When passing `NN_GX_MEM_VRAMA` or `NN_GX_MEM_VRAMB`, the memory region is allocated from VRAM-A or VRAM-B, respectively. Call the `nngxGetVramStartAddr()` function to get the starting address and the `nngxGetVramSize()` function to get the size, passing `NN_GX_MEM_VRAMA` or `NN_GX_MEM_VRAMB` as appropriate.

### Second Argument (Memory Region Use)

The second argument specifies the memory region use, passed as a `GLenum` type. The memory region byte alignment depends on the value passed.

When passing `NN_GX_MEM_TEXTURE`, the memory region is used for texture image data. The region is 128-byte aligned regardless of the data format.

When passing `NN_GX_MEM_VERTEXBUFFER`, the memory region is used for vertex buffers. Depending on the data stored, the memory may be 1-, 2-, or 4-byte aligned, but because the data type for storage is not passed to the allocator, we recommend an implementation that allocates at the maximum 4-byte alignment.

When passing `NN_GX_MEM_RENDERBUFFER`, the memory region is used for render buffers (color, depth, and stencil). The alignment may be 32-, 64-, or 96-byte aligned depending on the bits per pixel (16, 24, or 32), but again the format is not passed to the allocator. Consequently, either use a fixed format for the render buffer used by the application, or use an implementation that allocates at 192-byte alignment, which is the least common multiple.

When passing NN_GX_MEM_DISPLAYBUFFER, the memory region is used for display buffers. The region is 16-byte aligned regardless of the data format. **When allocating in VRAM, do not allocate the last 1.5 MB.**

When passing NN_GX_MEM_COMMANDBUFFER, the memory region is used for command lists. The region is 16-byte aligned.

When passing NN_GX_MEM_SYSTEM, the memory region is used for library system memory. Depending on the allocation size, the memory may be 1-, 2-, or 4-byte aligned, but to simplify matters, we recommend an implementation that allocates at the maximum 4-byte alignment.

#### Third Argument (Object Name)

The third argument specifies the name (ID) of the object, passed as a GLuint type. It is passed when the second argument is a value other than NN_GX_MEM_SYSTEM and is used when managing memory.

#### Fourth Argument (Memory Region Size)

The fourth argument specifies the memory region size, passed as a GLsizei type. Allocate memory of the specified size.

The application allocates memory of the appropriate alignment and size, and passes the starting address to the library as a void* type. Memory management must be handled by the application, such as remembering these four arguments and the starting address of the memory region as a set, and releasing memory using the deallocator function described later.

### 5.5.1.2. Deallocator

The deallocator function specified in the call to nngxInitialize is called to release the memory allocated by the allocator. Four arguments are passed to the deallocator. The first three have the same values as the first three arguments passed to the allocator function, and the last argument specifies the starting address of the memory region to release.

The application must take the argument values to identify the memory region allocated by the allocator and release it.

## 5.5.2. Command-List Object Creation

After the call to nngxInitialize() completes, the command-list objects required to run the gl() and nngx() library functions called for graphics processing must be created.

Create a command-list object using the nngxGenCmdlists() function, specify the current command list using the nngxBindCmdlist() function, and then use the nngxCmdListStorage() function to allocate the memory for the 3D command buffer and for accumulating command requests.

**Code 5-22. Creating Command List Objects**

```
void CreateCmdList()
{
    nngxGenCmdlists(1, &m_CmdList);
    nngxBindCmdlist(m_CmdList);
    nngxCmdlistStorage(256*1024, 128);
    nngxSetCmdlistParameteri(NN_GX_CMDLIST_RUN_MODE, NN_GX_CMDLIST_SERIAL_RUN);
}
```

A 3D command buffer of 256 KB and one command list that can accumulate up to a maximum of 128 command requests are allocated in this code example. However, it is possible to allocate multiple command lists and use them by switching between them each frame. When doing so, note that the command lists must be executed in the order that 3D commands were accumulated.

## 5.5.3. Allocating Memory for the Display Buffer and Render Buffer

Allocate memory for the display buffer, which is used for displaying to the LCD, and for the render buffer, which is the render target.

To allocate a framebuffer, you must bind each of the render buffers (color, depth, stencil) to a framebuffer object. During rendering, color data is written to the color buffer, depth data is written to the depth buffer, and stencil data is written to the stencil buffer. Note that the stencil buffer must share a buffer with the depth buffer.

If the format is the same for the upper and lower screens and there is no need for rendering in parallel, we recommend sharing the same framebuffer object and render buffer between the upper and lower screens to save memory. When doing so, set the buffer width and height large enough to accommodate both screens.

**Code 5-23. Allocating Render Buffers**

```
void CreateRenderbuffers(
        GLenum format, GLsizei width, GLsizei height)
{
    glGenFramebuffers(1, m_FrameBufferObject);
    glGenRenderbuffers(2, m_RenderBuffer);

    glBindRenderbuffer(GL_RENDERBUFFER, m_RenderBuffer[0]);
    glRenderbufferStorage(GL_RENDERBUFFER | NN_GX_MEM_VRAMA, format,
        width, height);
    glBindFramebuffer(GL_FRAMEBUFFER, m_FrameBufferObject);
    glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
        GL_RENDERBUFFER, m_RenderBuffer[0]);
    glBindRenderbuffer(GL_RENDERBUFFER, m_RenderBuffer[1]);
    glRenderbufferStorage(GL_RENDERBUFFER | NN_GX_MEM_VRAMB,
        GL_DEPTH24_STENCIL8_EXT, width, height);
    glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_STENCIL_ATTACHMENT,
        GL_RENDERBUFFER, m_RenderBuffer[1]);
}
```

This code sample allocates the color buffer in VRAM-A and the depth/stencil buffer in VRAM-B.

Under the usual framebuffer architecture, you can display the contents of the color buffer as is, but the 3DS color buffer data format is a block format and cannot be displayed on the LCD without first converting to a linear format. Consequently, in the 3DS system, the display buffer is between the color buffer and the actual display on the LCD. The contents of the color buffer are copied to the display buffer and then converted. You can also apply hardware-based anti-aliasing and vertical flipping during this process.

Initialization only allocates memory for the display buffer.

**Code 5-24. Allocating Display Buffers**

```
void CreateDisplaybuffers(
        GLenum format0, GLsizei width0, GLsizei height0, GLenum area0,
        GLenum format1, GLsizei width1, GLsizei height1, GLenum area1)
{
    // Upper Screen (DISPLAY0)
    nngxActiveDisplay(NN_GX_DISPLAY0);
    nngxGenDisplaybuffers(2, m_Display0Buffers);
    nngxBindDisplaybuffer(m_Display0Buffers[0]);
    nngxDisplaybufferStorage(format0, width0, height0, area0);
    nngxBindDisplaybuffer(m_Display0Buffers[1]);
    nngxDisplaybufferStorage(format0, width0, height0, area0);
    nngxDisplayEnv(0, 0);
    // Lower Screen (DISPLAY1)
    nngxActiveDisplay(NN_GX_DISPLAY1);
    nngxGenDisplaybuffers(2, m_Display1Buffers);
    nngxBindDisplaybuffer(m_Display1Buffers[0]);
    nngxDisplaybufferStorage(format1, width1, height1, area1);
    nngxBindDisplaybuffer(m_Display1Buffers[1]);
    nngxDisplaybufferStorage(format1, width1, height1, area1);
    nngxDisplayEnv(0, 0);
}
```

This code example uses two display buffers per LCD for multi-buffering. Display buffers are the only buffers where multiples must be allocated for multi-buffering.

The display buffers may be allocated from main memory (device memory). If you use a display buffer format that requires more bits per pixel than the format for the color buffer, copying from the color buffer causes an error.

This is the extent of initialization required for display to the LCD screens. At this point, run `nngxRunCmdlist` once to be sure that each buffer has been allocated.

## 5.6. Memory Management

Heap memory, device memory, and VRAM allocated for the application must be managed by the application itself. Classes are provided by CTR-SDK for memory management by applications.

### 5.6.1. Memory Blocks

The memory block feature is equivalent to the concept of an *arena* used with the NTR/TWL and Revolution systems. After a memory region of a particular size is allocated, memory can be cut from the memory block using the heap classes of the FND library (described later) and used. Instances of the heap classes defined in the FND library can be created by specifying a memory block as an argument.

Memory blocks are allocated in units of 4096 bytes. Memory blocks are generally used for allocating work memory or stacks from the heap memory available to applications, when allocations from device memory cannot be used. There are also libraries for simplifying the implementation of applications through the use of memory blocks.

Memory block allocation is handled by creating an instance of the `nn::os::MemoryBlock` class, or `nn::os::StackMemoryBlock` or `nn::os::StackMemory` class for stacks. Before creating an instance of these classes, you must call the `nn::os::InitializeMemoryBlock()` function to specify the memory region to use as a memory block.

**Code 5-25. Specifying a Memory Region to Use as a Memory Block**

```
void nn::os::InitializeMemoryBlock(uptr begin, size_t size);
```

Specify the start address of the memory region in *begin*, and the size of the memory region in *size*. Both must be aligned to nn::os::MEMORY_BLOCK_UNITSIZE (4096 bytes)

Either create an instance of the nn::os::MemoryBlock class (nn::os::StackMemoryBlock class for stacks) using a memory region of the size to be allocated for the memory block, or allocate the memory block using the Initialize() function after creating an empty instance. Size specifications must be made in units of 4096 bytes.

You can get the start address of the allocated memory block using the GetAddress() function and the size using the GetSize() function. In addition, the GetStackBottom and GetStackSize interface functions are provided for stacks so that you can directly pass stacks to threads. The SetReadOnly and IsReadOnly() functions can be used to get and set read-only attributes. However, there are no versions of these two functions for use with stacks.

When a memory block is no longer needed, it can be explicitly deallocated by calling the Finalize() member function.

> **Note:** The nn::os::SetupHeapForMemoryBlock() function remains because it is used in some sample demos. However, we recommend *not* calling it when using memory blocks.
>
> We recommend *not* using the nn::os::MemoryBlock and nn::os::StackMemoryBlock classes in your application, because future updates could potentially change the classes in a way that breaks your code.

## 5.6.2. Frame Heaps

A frame heap is a memory management class used to cut a memory region of a specified size out of a larger memory region specified at initialization time. In addition to using byte boundaries for the alignment specification, you can select to allocate memory from the beginning of the heap or the end of the heap depending on the sign used. Only when allocating memory from the beginning of the heap can you change the size of the memory region ultimately allocated.

The allocated memory regions cannot be deallocated individually. When deallocating memory, all allocated memory regions, or only the part allocated from the beginning, or only the part allocated from the end can be deallocated at the same time.

Instances of a frame heap can be specified using a class template (nn::fnd::FrameHeapTemplate) that locks operations, or with nn::fnd::FrameHeap, which does not lock operations, or with nn::fnd::ThreadSafeFrameHeap, which is thread-safe (lock operations are nn::os::CriticalSection).

## 5.6.3. Unit Heaps

A unit heap is a memory management class used to cut a fixed size memory region (called a *unit*) from a larger memory region specified at time of initialization. Alignment is specified at time of initialization, with 4-byte alignment specified by default. Consecutively allocated memory regions are not necessarily allocated from a continuous memory region.

Allocated memory regions can be deallocated independently. You can free all memory regions you have allocated by calling the `Invalidate()` function followed by the `Finalize()` function. This terminates the heap. To continue using the heap, first rebuild it by calling `Initialize`.

Instances of a unit heap can be created using a class template (`nn::fnd::UnitHeapTemplate`) that locks operations, or with `nn::fnd::UnitHeap`, which does not lock operations, or with `nn::fnd::ThreadSafeUnitHeap`, which is thread-safe (lock operations are `nn::os::CriticalSection`).

### 5.6.4. Expanded Heaps

An expanded heap is a memory management class used to cut a memory region of a specified size from a larger memory region specified at time of initialization. In addition to using byte boundaries for the alignment specification, you can select to allocate empty memory by searching from the beginning of the heap or searching from the end of the heap depending on the sign used. You can also change the size of the allocated memory region.

Allocated memory regions can be deallocated independently. If you repeatedly allocate and deallocate memory, there is a possibility that it will become impossible to allocate a memory region even though its size is smaller than the available memory obtained by the `GetTotalFreeSize()` function. This problem occurs because there is no longer a contiguous memory region of the specified size in the heap. You can get the maximum size that can be allocated as contiguous memory by using the `GetAllocatableSize()` function.

Instances of an expanded heap can be created using a class template (`nn::fnd::ExpHeapTemplate`) that locks operations, or with `nn::fnd::ExpHeap`, which does not lock operations, or with `nn::fnd::ThreadSafeExpHeap`, which is thread-safe (lock operations are `nn::os::CriticalSection`).

## 5.7. DLL Features (RO Library)

A dynamic link library (DLL) allows you to use a module that is loaded into memory dynamically. This reduces the amount of memory that an application uses for code and can make an application start up faster.

The CTR-SDK provides the RO library so that you can use DLLs.

### 5.7.1. DLL Glossary

This section explains the terminology related to the DLL features provided by the CTR-SDK. These terms may have different meanings and usage than they ordinarily would outside of the context of DLLs.

**Module**

A logical unit of executable code.

**Static module**

A module that has an entry function (`nnMain`) and is loaded at startup. You can make applications start faster by shrinking their static modules.

**Dynamic module**

A module that can be loaded and executed dynamically. You can reduce memory usage by splitting features into dynamic modules that are then switched in and out.

**Symbol**

Information that indicates the location of a variable or function in a module.

**Import (Reference)**

The act of using (or a statement that uses) a variable or function indicated by a symbol in another module.

**Resolve**

The act of making an imported symbol usable.

**Export (Publish)**

The act of allowing (or a statement that allows) other modules to use a variable or function indicated by a symbol.

**Export type**

The format in which a symbol is exported. There are three export types: names, indices, and offsets.

## 5.7.2. Characteristics and Limitations of the RO Library

The RO library has the following characteristics that set it apart from ordinary DLL implementations.

- Three different types of symbols can be exported.
  You can resolve a symbol by a name, index, or offset. Although it is possible to choose an export type for each individual symbol, the CTR-SDK build system only supports one export type per module.
  For more information about how to specify export types, see the *CTR-SDK Build System Manual (for DLLs)* and the *Guide to Developing a Build System (for DLLs)*.

- References between modules are resolved automatically.
  The simple act of loading a module resolves inter-module references so that functions and variables can be used. You can manually get a pointer to any symbol that has been exported as

a name or an index.

- Applications load modules into memory.
  Applications must load and adjust the location of dynamic modules in memory.

- C++ code is supported.
  You can use C++ code in a dynamic module. You can also import and resolve C++ symbols between modules. If you export a symbol as a name, however, you must use the symbol's mangled name to manually get a pointer to it.

- C++ exceptions can be caught between modules.
  Exceptions can be thrown by one module and then caught by another.

The RO library has the following restrictions and limitations.

- A single module cannot export more than 65,535 symbol names.
- The maximum length of a symbol that can be exported by name is 8,192 characters.
- You cannot include functions from the C standard library in dynamic modules.
- Alignment is ignored for any static variables or functions defined in a dynamic module with more than 8-byte alignment.
- Up to 64 dynamic modules can be loaded simultaneously.
- When using a nonstandard feature with weak symbols in C/C++, the system does not always operate as intended.

When using the static library as a dynamic module, first confirm with the creator of the library whether such use is possible.

**Warning:** Using the static libraries provided in the SDK as dynamic modules is prohibited.

Do not combine multiple dynamic modules in the same static library. This could cause various bugs to occur due to unintended library operation, including the existence of multiple global variables.

## 5.7.3. Files Used by the RO Library

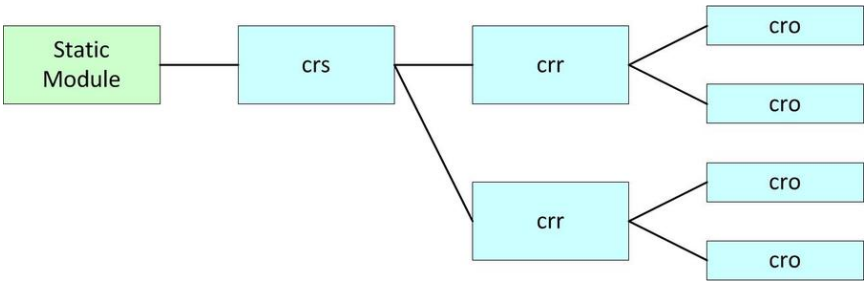
The RO library implements DLL features using data created with the following file formats.

**Table 5-8. Files Used by the RO Library**

| File Format (Extension) | Description |
|---|---|
| CRS | This file has import and export information for a static module. It does not contain executable code. Because an application can only have a single static module, it also only has one CRS file. |
| CRR | This file has management data for one or more dynamic modules. It must be placed directly under a specific directory (`/.crr/`) within a ROM archive. |
| CRO | This file has import and export information and executable code for a dynamic module. |

The following figure shows the relationship between these files.

**Figure 5-4. Relationship Between Files Used by the RO Library**




In general, you only need to create and use a single CRR file with information about all of the CRO files used by an application. Use multiple CRR files if you also want to optimize the amount of memory used by each CRR file or if you want to distribute additional programs.

> **Note:** For more information about how to create these files, see the *CTR-SDK Build System Manual (for DLLs)* or the *Guide to Developing a Build System (for DLLs)*.

## 5.7.4. Importing and Exporting

If two modules share header files, they can call each other's functions and access each other's variables with ordinary source code. They do not need to keep track of whether the symbols are exported.

If two modules do *not* share header files, they must use explicit import and export statements to call each other's functions and access each other's variables.

To import and export symbols explicitly, add the definitions in the following table to your source code.

**Table 5-9. Definitions for Explicit Import and Export Statements**

| Definition | Description |
|---|---|
| `NN_DLL_EXPORT` | Add this to function and variable declarations that you want to export to other modules. |
| `NN_DLL_IMPORT` | Add this to function and variable declarations that have been exported by other modules. |

**Code 5-26. Sample Code for Explicit Import and Export Declarations**

```
// Explicit export declarations
NN_DLL_EXPORT int g_Variable;
NN_DLL_EXPORT void Function()
{
    // (omitted)
}

// Explicit import declarations
extern NN_DLL_IMPORT int g_Variable;
extern NN_DLL_IMPORT void Function();
```

**Note:** If you export a symbol explicitly, it is always exported. It will not be dead-stripped, even if it is not used within a module.

## 5.7.5. A Comparison of Export Types

The following table shows the differences caused by symbol export types.

**Table 5-10. Differences Caused by Export Types**

| Item | Name | Index | Offset |
|---|---|---|---|
| What is the size of the import information? | Large | Small | Small |
| What is the size of the export information? | Large | Small | 0 |
| What is a module's required load speed? | Slow | Average | Fast |
| Can pointers be obtained manually? | Yes | Yes | No |
| Can the imported module be created in advance? | Yes | No | No |
| How many build steps? | Normal | Normal | Many |

Symbol names present richer functionality than symbol indices, which in turn present richer functionality than offsets. However, functionality does entail other costs in file size and processing time.

In general, choose to export symbols as offsets. If you want to get symbol pointers manually or use dynamic modules like libraries, choose to export symbols as names or indices as necessary.

**Note:** Public types can also be specified for symbols in static modules.

## 5.7.6. Special Dynamic Module Functions

If a dynamic module's source code exports functions with particular names, the RO library calls those functions when the module is initialized and at other specified times.

**Code 5-27. Special Functions**

```
extern "C" NN_DLL_EXPORT void nnroProlog();
extern "C" NN_DLL_EXPORT void nnroEpilog();
extern "C" NN_DLL_EXPORT void nnroUnresolved();
```

The `nnroProlog()` function is called after the `DoInitialize()` function has finished initializing a module. This allows you to implement module-specific initialization.

The `nnroEpilog()` function is called after the `DoFinalize()` function has finished finalizing a module. This allows you to implement module-specific finalization.

The `nnroUnresolved()` function is called when a module uses an external symbol that has not been resolved. This allows you to detect and handle calls to unresolved symbols.

Implement these functions in your application as necessary.

**Note:** The `nnroUnresolved()` function may also be used in a static module. However, one restriction is that it will not work if a symbol whose reference has been resolved later becomes unresolved.

## 5.7.7. Basic Workflow

This section explains the basic workflow for using a dynamic module.

### 5.7.7.1. Initialize the RO Library

Initialize the RO library with the `nn::ro::Initialize()` function. Initialization requires a CRS file, which must already be loaded into memory. Note that the content of the CRS file must be stored in a buffer that has been allocated in a memory region outside of device memory, with a starting address that is `nn::ro::RS_ALIGNMENT` (4,096 bytes) aligned and the size a multiple of `nn::ro::RS_UNITSIZE` (4,096 bytes).

The RO library is responsible for managing the buffer with the CRS file. Do not overwrite or release this buffer until you have finalized the RO library.

### 5.7.7.2. Register Management Data

Before using a dynamic module, you must load the CRR file that manages it and then register management data with the `nn::ro::RegisterList()` function. Note that the content of the CRR file must be stored in a buffer that has been allocated in a memory region outside of device memory, with a starting address that is `nn::ro::RR_ALIGNMENT` (4,096 bytes) aligned and the size a multiple of `nn::ro::RR_UNITSIZE` (4,096 bytes).

The RO library is responsible for managing the buffer with the CRR file. Do not overwrite or release this buffer until you have unregistered the management data by calling the `Unregister()` function on the `nn::ro::RegistrationList` class pointer returned by the `nn::ro::RegisterList()` function.

### 5.7.7.3. Load Dynamic Modules

Load dynamic modules with the `nn::ro::LoadModule()` function. The application must load a module's CRO file into memory in advance. Note that the content of the CRO file must be stored in a buffer that has been allocated in a memory region outside of device memory, with a starting address that is `nn::ro::RO_ALIGNMENT_LOAD_MODULE` (4,096 bytes) aligned and the size a multiple of `nn::ro::RO_UNITSIZE_LOAD_MODULE` (4,096 bytes).

You must also provide a buffer to be used for the `.data`/`.bss` sections. This buffer must have a starting address that is `nn::ro::BUFFER_ALIGNMENT` (8 bytes) aligned. It must not be smaller than the value of the `bufferSize` member variable in the `nn::ro::SizeInfo` structure that is obtained when the content of the CRO file (`REQUIRED_SIZE_FOR_GET_SIZE_INFO` bytes from the beginning) is passed to the `nn::ro::GetSizeInfo()` function. The `SizeInfo` structure

also contains information about memory regions that can be used after modules are loaded. If `bufferSize` is smaller than the memory region that is freed after a module is loaded, the buffer can be allocated from that memory region.

The following table shows how the value specified for *`fixLevel`* affects dynamic module features and the memory regions that are released after modules are loaded. If nothing is specified for *`fixLevel`*, it is treated as if it were `FIX_LEVEL_1`.

**Table 5-11. Effects of fixLevel on Dynamic Module Behavior**

| Item | `FIX_LEVEL_0` | `FIX_LEVEL_1` | `FIX_LEVEL_2` | `FIX_LEVEL_3` |
|---|---|---|---|---|
| Can this module be reused after it is unloaded? | Yes | No | No | No |
| When another module is loaded, are its symbols automatically linked with this module? | Yes | Yes | No | No |
| When another module is loaded, are this module's symbols automatically linked with that module? | Yes | Yes | Yes | No |
| Can you get the address of the symbols in this module? | Yes | Yes | Yes | No |
| Are the symbols in all modules that have been loaded until now automatically linked with this module? | Yes | Yes | Yes | Yes |
| What is the address at the end of the memory region managed by the library? | `fix0End` | `fix1End` | `fix2End` | `fix3End` |

A memory region is released after a module is loaded. The size of the region for `FIX_LEVEL_2` is less than or equal to that of `FIX_LEVEL_3`, the region for `FIX_LEVEL_1` is smaller, and the region for `FIX_LEVEL_0` is smaller still. More memory is released when symbol names are exported than when any other type of symbol is exported.

In general, specify `true` for *`doRegister`* so that any references to unresolved symbols are automatically resolved when another dynamic module is loaded. If you specify `false`, references are resolved to symbols in dynamic modules that have already been loaded but they are not automatically resolved when another module is loaded. In the latter case, you must manually resolve symbols by calling `Link` with the `nn::ro::Module` pointer returned by the `nn::ro::LoadModule()` function.

## 5.7.7.4. Start to Use Dynamic Modules

Before you use the functions and variables in a dynamic module, you must call `DoInitialize` with the `nn::ro::Module` pointer returned by the `nn::ro::LoadModule()` function. This constructs global objects in the dynamic module and runs the initialization process implemented by the `nnroProlog()` function.

You can call the `DoInitialize()` function immediately after a module is loaded as long as its initialization process does not reference global objects in another dynamic module. If the initialization processes for multiple dynamic modules reference *each other's* global objects, however, load all of those dynamic modules *before* calling the `DoInitialize()` function.

You can use the `Module` class's `GetName()` function to get the name of a dynamic module. This function returns the same name that was specified when the module was built. (If you are using the CTR-SDK build system, this is the name specified by `TARGET_MODULE`). This name is also used by the `nn::ro::FindModule()` function to search for loaded dynamic modules.

You can use the `Module::IsAllSymbolResolved()` function to determine whether all external references from a dynamic module have been resolved. To unresolve all of a module's resolved references, including references to the module itself, call the `Module::Unlink()` function. To resolve those references again, call the `Module::Link()` function.

There are two ways to manually get a pointer to a symbol: the `nn::ro::GetPointer()` function, which searches through the symbols in every module for a name, and the `nn::ro::Module::GetPointer()` function, which searches through the symbols in a particular module for a name or index. You can also use the `nn::ro::GetAddress()` function to get the address of a symbol that you found by name.

## 5.7.7.5. Finalize Dynamic Modules

To free the memory region used by a module that is no longer necessary, you must first call the `Module` class's `DoFinalize()` function to destroy the global objects in that dynamic module and run the shutdown processing implemented by the `nnroEpilog()` function. After this is complete, call the `Module` class's `Unload()` function to unload the dynamic module.

When a dynamic module is unloaded, it returns to its preloaded state: any memory managed by the library is released and all references are made unresolved. You can then release the buffer that had the content of the CRO file.

You generally need to reload a dynamic module if you want to reuse it after it has been unloaded. However, if you loaded the module with `FIX_LEVEL_0` (`FIX_LEVEL_NONE`) specified as *`fixLevel`*, you can call the `nn::ro::LoadModule()` function to reuse the same buffer and other settings without reloading the CRO file. To reuse the buffer, you must have not initialized any static variables or you must have initialized all static variables with the `nnroProlog()()` function There are restrictions on the use of static variables because if they were overwritten while the dynamic module was in use, they are not restored to their preloaded state when the module is unloaded.

## 5.7.7.6. Unregister Management Data

To unregister the management data and release the memory region, you must call the `Unregister()` function with the `nn::ro::RegistrationList` class pointer returned by the `nn::ro::RegisterList()` function.

After the management data has been unregistered, the buffer with the content of the CRR file is no longer managed by the library. Even though you cannot load a new dynamic module managed by that CRR file after the management data has been unregistered, you can still use dynamic modules that have already been loaded.

## 5.7.7.7. Finalize the RO Library

Finalize the RO library with the `nn::ro::Finalize()` function.

After you have called the `nn::ro::Finalize()` function, you can release all the memory that was used by the RO library; this includes the buffers with the content of the CRS, CRR, and CRO files.

### 5.7.8. Enumeration and Searching of Dynamic Modules

All dynamic modules can be enumerated, regardless of whether they are loaded by automatic link with the `nn::ro::Module::Enumerate()` function. Pass the class inherited from the `nn::ro::Module::EnumerateCallback()` function to the argument of the `Enumerate()` function. For every dynamic module, the `operator()` function is called back with the pointer to the dynamic module as an argument.

Dynamic modules can be searched by the `nn::ro::Module::Find()` function. If `nn::ro::Module::Find` finds the dynamic module whose specified string and name match, it returns the pointer to the module, and if it cannot find it, it returns `NULL`. However, make sure that the search target is only a dynamic module that is loaded by automatic link.

### 5.7.9. Information About the Memory Region Used by Dynamic Modules

Information about the memory region used by dynamic modules can be obtained by the `nn::ro::Module::GetRegionInfo()` function. The information of the memory region is stored in the `nn::ro::RegionInfo` structure passed to the *`pri`* argument.



# 6. Input Devices

Use the libraries and classes that support each input device to easily incorporate into your application the input from the various input devices on the system, such as the digital buttons, Circle Pad, touch panel, accelerometer, gyro sensor, microphone, and cameras.

## 6.1. Libraries That Use Devices

To access the input and other devices on the system from your application, you must use the libraries provided by CTR-SDK. The functions in many of the libraries for using devices return an instance of the `nn::Result` class.

Call the `nn::Result` class member function `IsSuccess` or `IsFailure` to find out whether the function completed processing successfully. If `IsSuccess` returns `true`, the function completed successfully; if it returns `false`, the function failed. The `IsFailure` return values have the opposite meaning; where `true` means the function failed, and `false` means the function completed

successfully.

The `nn::Result` class records error severity, a description, and the name of the module that raised the error. In addition to checking for function success, you may sometimes need to use this detailed information to find a way to work around or fix an error in your code.

## 6.2. HID Library

Using the HID library, you can handle input from the digital buttons (+Control Pad, A/B/X/Y/L/R Buttons, and START), Circle Pad, touch panel, accelerometer, gyro sensor, debug pad, and Circle Pad Pro.

Call the `nn::hid:Initialize()` function to initialize the HID library. Successive calls to the initialization function do nothing and return an error.

After initialization, input is sampled automatically, and you can obtain input from each device based on the corresponding class. The timing and cycle of the input sampling start differ depending on the type of device. To end sampling, the class created by the acceleration and gyro sensors is destroyed. Call the `nn::hid::ExtraPad::StopSampling()` function when done to finalize the Circle Pad Pro, and for all others, finalize the HID library.

> **Note:** The C Stick, ZL Button, and ZR Button on SNAKE hardware can be handled the same way as a CTR unit with the Circle Pad Pro always attached. In this case, the C Stick corresponds to the Right Circle Pad, and the ZL and ZR Buttons correspond to the same buttons on the Circle Pad Pro.
>
> For information about the C Stick and differences with the Circle Pad Pro, see 6.2.7. C Stick.

**Table 6-1. Types and Inputs**

| Type | Class Handling Input | Sampling Start Trigger | Cycle |
|---|---|---|---|
| Digital buttons and Circle Pad | `nn::hid::PadReader` | HID library initialization | 4 ms |
| Touch panel | `nn::hid::TouchPanelReader` | HID library initialization | 4 ms |
| Accelerometer | `nn::hid::AccelerometerReader` | Reader class creation | Avg. 10 ms |
| Gyro sensor | `nn::hid::GyroscopeReader` | Reader class creation | Avg. 10 ms |
| Debug pad | `nn::hid::DebugPadReader` | HID library initialization (when connected) | 16 ms |
| Circle Pad Pro | `nn::hid::ExtraPadReader` | `ExtraPad` class `StartSampling()` function | 8–32 ms |
| C Stick, ZL / ZR Button | `nn::hid::ExtraPadReader` | `ExtraPad` class `StartSampling()` function | 8–32 ms |

> **Note:** The library allows you to set a sampling frequency of 8 to 32 milliseconds for the C Stick, ZL Button, and ZR Button on SNAKE hardware, but we recommend setting a sampling frequency of 10 to 21 milliseconds to match the hardware capabilities.

Call the nn::hid::Finalize() function to finalize use of the library. When using the nn::hid::ExtraPadReader class, you must first call the nn::hid::ExtraPad::Finalize() function.

## 6.2.1. Digital Buttons and Circle Pad

Call the nn::hid::PadReader class's Read or ReadLatest member functions to get input from the digital buttons and Circle Pad as an nn::hid::PadStatus structure. The Read() function can get sampling results in order of the latest result, but it cannot reacquire sampling results. When the function is called at a cycle faster than the sampling cycle, it cannot get sampling results. On the other hand, the ReadLatest() function can get only the latest sampling result. It can also get sampling results again, so even when it is called at a cycle faster than the sampling cycle, it can still get sampling results.

The nn::hid::PadStatus structure's hold member variable records the button being held when input was sampled, the trigger variable records the button that was pressed when input was sampled, and release variable records the button that was released when input was sampled, all mapped to bit values. For the ReadLatest() function, both the trigger and release members are evaluated for their states at that time, so any changes between calls are not necessarily applied. Input bits for +Control Pad input may in fact be input from the Circle Pad emulating the +Control Pad. The actual Circle Pad input is recorded to the stick member variable as a biaxial coordinate value.

**Table 6-2. Buttons and Definitions**

| Definition | Corresponding Button |
|---|---|
| BUTTON_UP | +Control Pad (Up) |
| BUTTON_DOWN | +Control Pad (Down) |
| BUTTON_LEFT | +Control Pad (Left) |
| BUTTON_RIGHT | +Control Pad (Right) |
| BUTTON_A | A Button |
| BUTTON_B | B Button |
| BUTTON_X | X Button |
| BUTTON_Y | Y Button |
| BUTTON_L | L Button |
| BUTTON_R | R Button |
| BUTTON_START | START or SELECT (for standard operations) |
| BUTTON_SELECT_FOR_DEBUGGING | SELECT (only for debugging) |
| BUTTON_EMULATION_UP | Emulated +Control Pad (Up) input using Circle Pad |
| BUTTON_EMULATION_DOWN | Emulated +Control Pad (Down) input using Circle Pad |
| BUTTON_EMULATION_LEFT | Emulated +Control Pad (Left) input using Circle Pad |
| BUTTON_EMULATION_RIGHT | Emulated +Control Pad (Right) input using Circle Pad |

Use the nn::hid::EnableSelectButton() and nn::hid::DisableSelectButton() functions to enable and disable SELECT sampling.

**Code 6-1. Enabling and Disabling SELECT Sampling**

```
bool nn::hid::EnableSelectButton();
void nn::hid::DisableSelectButton();
```

Calling the `EnableSelectButton()` function returns `true` when the function has successfully enabled sampling of SELECT. The function always returns `false` and the sampling cannot be enabled unless the system is set to debug mode. Calling the function when SELECT sampling is already enabled returns `BUTTON_SELECT_FOR_DEBUGGING`, so you can distinguish the SELECT input from the START input.

The Circle Pad may return a coordinate value even when the user is not touching it. Consequently, make sure that you set an appropriate play value and clamp mode with the `nn::hid::PadReader` class's `SetStickClamp()` and `SetStickClampMode()` member functions, respectively. There are three clamp modes available: circular (`STICK_CLAMP_MODE_CIRCLE`), cruciform (`STICK_CLAMP_MODE_CROSS`), and minimal (`STICK_CLAMP_MODE_MINIMUM`). Use the `GetStickClampMode()` function to get the current clamp mode, and use the `GetStickClamp()` function to get the clamp value. Circular clamping is the default clamp setting. The following examples show clamped Circle Pad input, with the original input value coordinates specified as `(x, y)`, the distance from the origin as `d`, the clamp minimum and maximum values as min and max, the clamped coordinates as `(x', y')`, and the clamped distance from the origin as `d'`.

### Circular Clamping

d <= min

(x', y') = (0, 0)

min < d < max

(x', y') = ((d - min) / d * x, (d - min) / d * y)

d > max

(x', y') = ((max - min) / d * x, (max - min) / d * y)

### Cruciform Clamping

x < 0

(x', y') = (x + min, y) But x + min must be 0 or greater.

x >= 0

(x', y') = (x - min, y) But x - min must be 0 or less.

y < 0

(x', y') = (x, y + min) But y + min must be 0 or greater.

y >= 0

(x', y') = (x, y - min) But y - min must be 0 or less.

d' > (max - min)

(x', y') = ((max - min) / d * x, (max - min) / d * y)

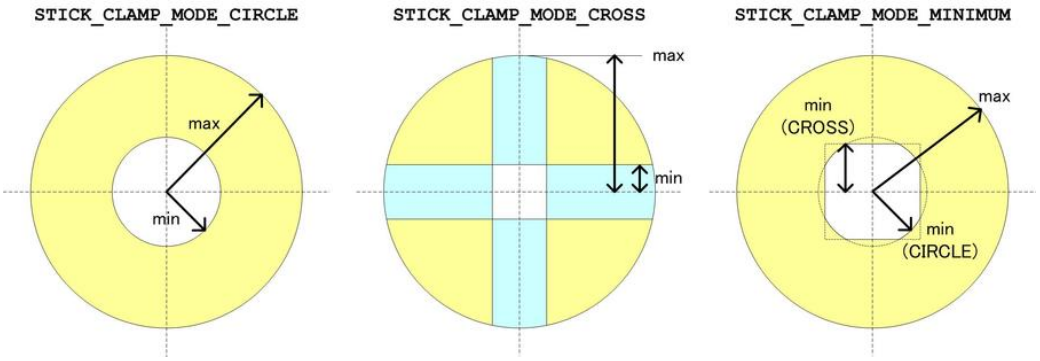
### Minimal Clamping

Minimal clamping combines circular and cruciform clamping. When using minimal clamping (minimal values), unclamped coordinate values are those that lie within the unclamped ranges of both the circular and cruciform clamping ranges. Coordinates outside this region are clamped to within this

minimum.

Figure 6-1 shows how the ranges for clamped Circle Pad input coordinates change.

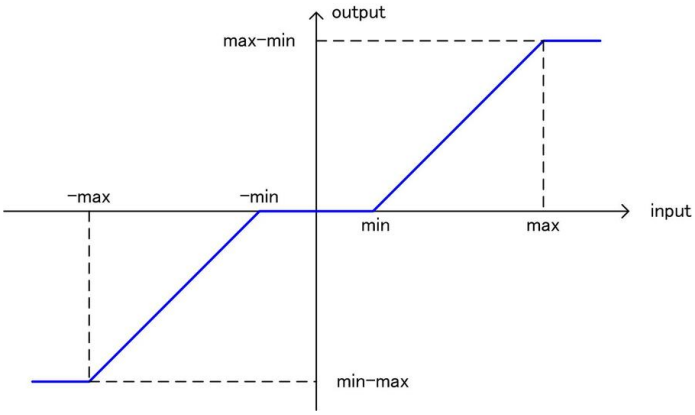
**Figure 6-1. Valid Input Ranges for Circular, Cruciform, and Minimal Clamping**




In `STICK_CLAMP_MODE_CROSS`, for coordinates close to either the x-axis or y-axis, only the value for that axis changes (that is, if close to the x-axis, only the x-coordinate value changes).

Clamped input coordinate values for either axis produce output values as shown below in any of the clamp modes.

**Figure 6-2. Relationship Between Input and Output Coordinates**




In any of the clamp modes, the output value is `0` until the input value passes the `min` threshold, and the output value is `±(max − min)` after the input value passes the `max` threshold, with the clamped output value ranging from `0` to `±(max − min)`.

In any of the clamp modes, the obtainable output coordinates describe a circle with radius `(max − min)`. Choose the appropriate clamp mode for the input you need. `STICK_CLAMP_MODE_CIRCLE` preserves the angle of the input coordinates from the origin and is better suited to uses requiring finer directional control. `STICK_CLAMP_MODE_CROSS` only outputs the coordinate value for the closest axis, either the x-axis or the y-axis, and is better suited to uses emphasizing axial directional input, such as when selecting items from a menu.

## Normalizing Circle Pad Coordinate Values

The `nn::hid::PadReader` class's `NormalizeStick()` member function takes the Circle Pad's coordinate value obtained from the `Read()` or `ReadLatest()` member functions and normalizes it to an `f32` floating-point value between –1.0 and +1.0 with `(max − min) = 1.0`.

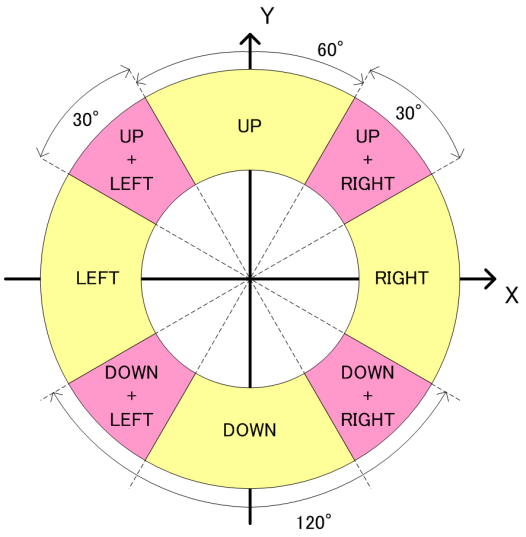
Use the `NormalizeStickWithScale()` function to normalize Circle Pad coordinate values with a sensitivity adjustment. Configure the sensitivity adjustment by calling the

`SetNormalizeStickScaleSetting()` function and specifying values for the *`scale`* (default of `1.5`) and *`threshold`* (default of `141`) parameters. Normalizing multiplies values that are lower than *`threshold`* by (1/*`scale`*), while values equal to *`threshold`* or higher gradually approach `±1.0`. Use this feature to make the Circle Pad respond as if its movable range were *`scale`* times big.

#### Emulating the +Control Pad on the Circle Pad

The +Control Pad emulation bits for Circle Pad input (`BUTTON_EMULATION_*`) are used to judge which direction is being indicated by the controller, by using circular clamping with a lower limit of 40 and an upper limit of 145. The values set using the `SetStickClamp()` and `SetStickClampMode()` functions are not used for this purpose. Positive and negative values for the x-axis and y-axis are interpreted to mean left, right, up, and down, respectively, and the bit for a particular direction is set if that direction is within 120° of the input direction. Consequently, the 30° overlap zone between directions is interpreted as a diagonal input.

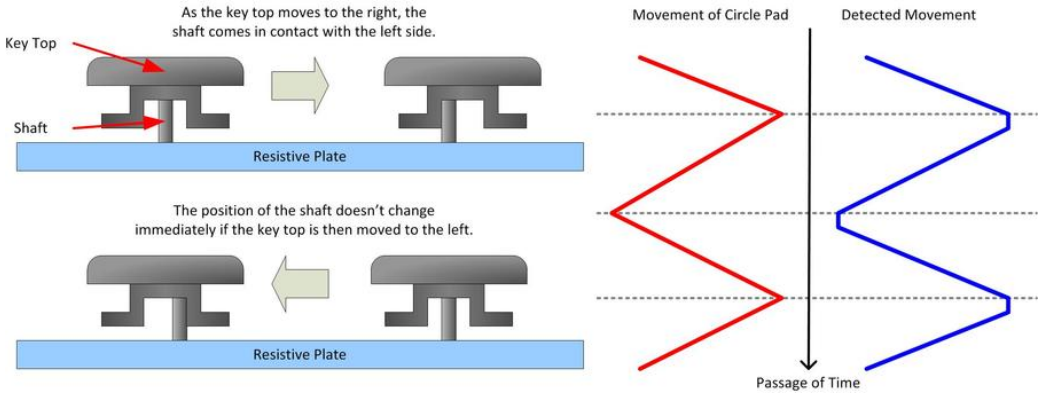
**Figure 6-3. BUTTON_EMULATION_ Judgment Range***




## 6.2.1.1. Circle Pad Hardware Performance

The Circle Pad included in the 3DS system is made up of a resistive plate, shaft, and key top. This composite structure includes some tolerance, resulting in a slight lag in response compared to the analog sticks for the Nintendo GameCube or Wii consoles when reversing the direction of input.

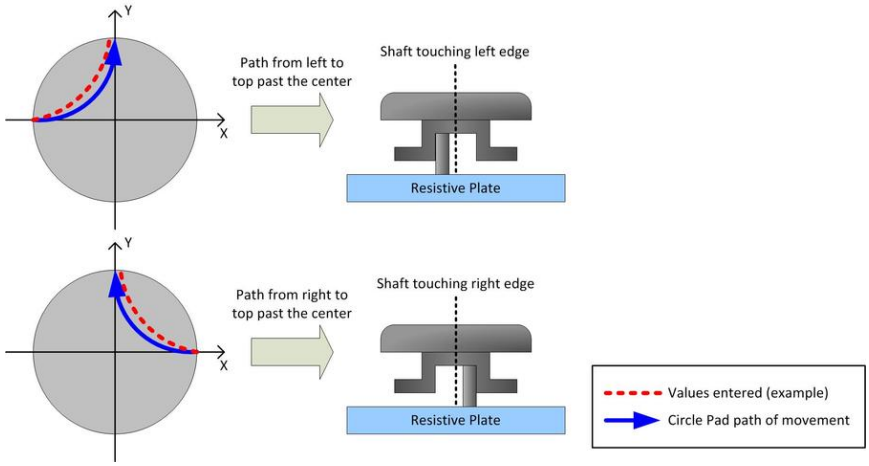
**Figure 6-4. Delayed Response to Input Inversion Due to the Circle Pad's Hardware**


As the key top moves to the right, the shaft comes in contact with the left side.
Key Top
Shaft
Resistive Plate
The position of the shaft doesn't change immediately if the key top is then moved to the left.
Resistive Plate
Movement of Circle Pad
Detected Movement
Passage of Time

Actual input values might also differ even when moving the Circle Pad along the same path of movement. This phenomenon is called path dependence and is also caused due to the Circle Pad hardware.

For example, when moving the Circle Pad from the left side to the top, going past the center, the shaft is touching the left edge of the key top, with input values tending more toward –X. Conversely, when moving the Circle Pad from the right side to the top, going past the center, the shaft is touching the right edge of the key top, with input values tending more toward +X. Input values exhibit this same tendency in the vertical direction for ±Y values.

**Figure 6-5. Circle Pad Path Dependence**




We recommend that your application factor in some tolerance for this phenomenon, such as for input detection thresholds.

## 6.2.2. Touch Panel

Call the `nn::hid::TouchPanelReader` class's `Read` or `ReadLatest` member functions to get input from the touch panel as an `nn::hid::TouchPanelStatus` structure. The `Read()` function can get sampling results in order of the latest result, but it cannot reacquire sampling results. When the function is called at a cycle faster than the sampling cycle, it cannot get sampling results. On the other hand, the `ReadLatest()` function can get only the latest sampling result. It can also get sampling results again, so even when it is called at a cycle faster than the sampling cycle, it can still get sampling results.

The `nn::hid::TouchPanelStatus` structure's `x` and `y` members record the touch panel input coordinates in pixels, with the upper-left corner as the origin (when holding the system so that the

lower screen is toward the user). Note that the coordinate axes on the touch panel differ from the LCD coordinate axes used in the GX library. Areas in the five-dot region on the outermost of the screen that is difficult to touch are returned as clamped coordinate values. Accordingly, the x values actually returned are between 5 and 314, and the y values between 5 and 234.

**Code 6-2. Converting Touch Panel Input Coordinates to LCD Coordinates**

```
x = nn::gx::DISPLAY1_WIDTH - touchPanel.y;
y = nn::gx::DISPLAY1_HEIGHT - touchPanel.x;
```

The `touch` member records whether the stylus is touching the touch panel. The member value is `0` if the stylus is not touching the panel and 1 if it is.

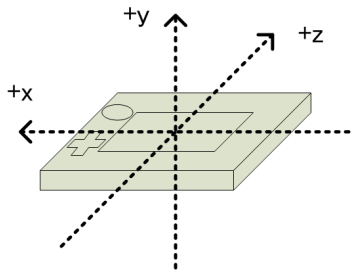
## 6.2.3. Accelerometer

Call the `Read` or `ReadLatest` member functions of the `nn::hid::AccelerometerReader` class to get input from the accelerometer as an `nn::hid::AccelerometerStatus` structure. The `Read()` function can get sampling results in order of the latest result, but it cannot reacquire sampling results. When the function is called at a cycle faster than the sampling cycle, it cannot get sampling results. On the other hand, the `ReadLatest()` function can get only the latest sampling result. It can also get sampling results again, so even when it is called at a cycle faster than the sampling cycle, it can still get sampling results.

The accelerometer is turned on when an instance of the `AccelerometerReader` class is created and it is turned off when an instance is destroyed. To conserve battery life, only instantiate the `AccelerometerReader` class when it is used. However, if you repeatedly create and destroy an instance of the `AccelerometerReader` class in a function that is called every frame (for example), the frequent power cycling wastes battery power and leads to unexpected malfunctions.

The `x`, `y`, and `z` members of the `nn::hid::AccelerometerStatus` structure record input from the accelerometer's three axes. The x-axis is equivalent to the left-right axis on the +Control Pad, the y-axis is equivalent to a line perpendicular to the lower LCD screen, and the z-axis is equivalent to the up-down axis on the +Control Pad. The raw input values from these three axes do not accurately describe the acceleration of the system, so make sure that you first convert the raw values to G units using the `nn::hid::AccelerometerReader` class's `ConvertToAcceleration()` member function before using them in your application.

**Figure 6-6. The Three Accelerometer Axes**




The accelerometer’s zero point may vary by up to 0.05 G over time and 0.08 G due to temperature changes, for a maximum variance of 0.13 G. Applications that use the accelerometer must account for this variance. Variance over time can be mostly rectified by calibrating, but variance due to temperature can reoccur in a short period even after calibrating. This 0.08 maximum temperature-induced variance equates to a roughly five-degree tilt. When detecting movements of finer

precision, we recommend calibrating immediately if the user notices anything strange.

> **Note:** Hold the Y or B Button for three seconds while the HOME Menu is displayed to calibrate the accelerometer.

The accelerometer's maximum sensitivity may also vary by up to ±8%. After factoring in the zero-point variance offset, the maximum input value is 10% less than the theoretical maximum value of approximately 1.8 G, at 1.62 G. Applications must not depend on any input values greater than this.

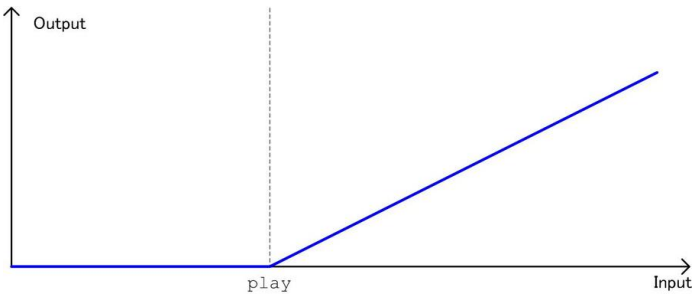
> **Warning:** The output value of the accelerometer has a superimposed at-rest noise of ±0.02 G. The output value has an additional superimposed noise of ±0.05 G from conductive vibration when the speaker is outputting sound. The effect of this noise input must be considered when detecting small acceleration.
>
> The conductive vibration is highest in the 1-kHz range. The effect can be reduced by adjusting the sound playback volume.

Call the `SetSensitivity` member function of the `nn::hid::AccelerometerReader` class to set the input play and detection sensitivity values, and use the `GetSensitivity` member function to get these values. Either of these values may be set in the range from 0 to `MAX_OF_ACCELEROMETER_SENSITIVITY`, with the default values of 0 for the play and `MAX_OF_ACCELEROMETER_SENSITIVITY` for the detection sensitivity. Setting the detection sensitivity to `0` means only `0` will be returned. Specifying `MAX_OF_ACCELEROMETER_SENSITIVITY` for this setting gives a response based directly on the value read from the device.

Looking at the accelerometer input value as a set of triaxial coordinates, the accelerometer change value for any axis is interpreted to be within the play range if it is no greater than the play value. Values within the play range are output as 0, and changes within this range are output as no change at all.

The following figure shows the relationship between the accelerometer input and output values, and the `play` range. Use this play range setting to minimize any false readings caused by small changes in position, such as being bumped by the user's hands.

**Figure 6-7. Relationship Between Input and Output Values Due to the Play Range Setting**




Use the `nn::hid::AccelerometerReader` class to carry out axial rotation on output values (already adjusted for sensitivity and play) after adjusting for an offset. You can also use axial rotation of output values to apply any angle tilt to the accelerometer.

**Code 6-3. Offsetting Accelerometer Output Values**

```
class nn::hid::AccelerometerReader
{
```

```
    void EnableOffset();
    void DisableOffset();
    bool IsEnableOffset() const;
    void SetOffset(s16 x, s16 y,s16 z);
    void GetOffset(s16* pX, s16* pY, s16* pZ) const;
    void ResetOffset();
}
```

Enable offsetting of output values by calling `EnableOffset`, and disable by calling `DisableOffset`. Call `IsEnableOffset` to get the current setting; a return value of `true` indicates that the offset is enabled. When the offset is enabled, each component of the offset value is subtracted from each component of the output value.

Set the offset value itself by calling `SetOffset`. Use `SetOffset` to specify offset values individually for each axis. Note that each component of the offset value is specified not as an acceleration value (in G), but rather as a pre-conversion sampling value. Use the `GetOffset()` function to get the current setting. Call `ResetOffset` to return the offset to the initial value of 0 for each component.

**Note:** The `SetOffsetFromBaseStatus()` function has been removed from CTR-SDK.

**Warning:** Do not use the offset value settings to calibrate the accelerometer.

**Code 6-4. Accelerometer Axial Rotation**

```
class nn::hid::AccelerometerReader
{
    void EnableAxisRotation();
    void DisableAxisRotation();
    bool IsEnableAxisRotation() const;
    void SetAxisRotationMatrix(const nn::math::MTX34 mtx);
    void GetAxisRotationMatrix(nn::math::MTX34* pMtx) const;
    void ResetAxisRotationMatrix();
}
```

Enable axial rotation of output values by calling `EnableAxisRotation`, and call `DisableAxisRotation` to disable. Rotation is disabled by default. Call `IsEnableAxisRotation` to get the current setting; a return value of `true` indicates that rotation is enabled. When enabled, the previously set rotation matrix is applied to the output value.

To set the rotation matrix (a 3×4 matrix), call `SetAxisRotationMatrix`; to get the currently set matrix, call `GetAxisRotationMatrix`. Call `ResetAxisRotationMatrix` to revert to the identity matrix (no rotation).

When both an offset rotation and axial rotation are enabled, the offset is applied first, and then the rotation is applied to the resulting value.

**Warning:** Do not use the axial rotation settings to calibrate the accelerometer.

## 6.2.3.1. Cautions for Implementing Application-Specific Calibration Routines

**Warning:**Applications are now prohibited from implementing their own calibration routines.

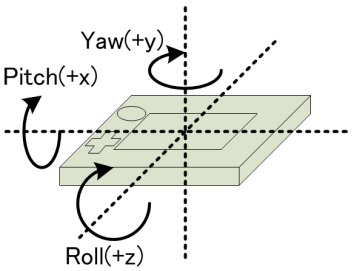
## 6.2.4. Gyro Sensor

Call the `nn::hid::GyroscopeReader` class's `Read` or `ReadLatest` member functions to get the gyro sensor's input value as a `nn::hid::GyroscopeStatus` structure. The `Read()` function can get sampling results in order of the latest result, but it cannot reacquire sampling results. When the function is called at a cycle faster than the sampling cycle, it cannot get sampling results. On the other hand, the `ReadLatest()` function can get only the latest sampling result. It can also get sampling results again, so even when it is called at a cycle faster than the sampling cycle, it can still get sampling results.

The gyro sensor is turned on when an instance of the `GyroscopeReader` class is created, and it is turned off when an instance is destroyed. To conserve battery life, only instantiate the `GyroscopeReader` class when it is used. However, if you repeatedly create and destroy an instance of the `GyroscopeReader` class in a function that is called every frame (for example), the frequent power cycling wastes battery power and leads to unexpected malfunctions.

The `speed`, `angle`, and `direction` members of the `nn::hid::GyroscopeStatus` structure record the gyro sensor's angular velocity, angle of rotation, and 3D attitude. The angular velocity and angle of rotation values are 3D vectors, with the x component recording the pitch direction, the y component the yaw direction, and the z component the roll direction. An angular velocity of 360 dps is represented by a value of `1.0`, and an angle of rotation of 360° by a value of `1.0`. Pitch is the degree of tilt forward and back about the axis running along the width of the lower screen, yaw is rotation about an axis perpendicular to the lower screen, and roll is tilt to left and right about the axis running along the height of the lower screen.

**Figure 6-8. The Three Components of the Gyro Sensor**




The gyro sensor's maximum sensitivity to angular velocity can vary by up to ±8%. After factoring in the zero-point variance offset, the maximum input value is 10% less than the theoretical maximum value of approximately 1800 dps, at 1620 dps. Applications must not depend on any input values greater than this.

The 3D attitude is recorded as a 3 × 3 matrix. This matrix is calculated based on the angular velocity, and you can change the angular velocity multiple referenced in calculation by calling the `nn::hid::GyroscopeReader::SetDirectionMagnification()` function. Specify a multiple of 1.0 to use the output angular velocity without modification in calculation. Specify 2.0 to use a value of double the detected angular velocity.

The `nn::hid::GyroscopeReader` class detects when the system is at rest and automatically calibrates the zero-point offset (zero-point drift correction). The correction setting is applied alike to all directions.

**Code 6-5. Gyro Sensor Zero-Point Drift Correction**

```
class nn::hid::GyroscopeReader
{
    void EnableZeroDrift();
    void DisableZeroDrift();
    bool IsEnableZeroDrift() const;
    f32 GetZeroDriftEffect() const;
    void ResetZeroDriftMode();
    void SetZeroDriftMode(const nn::hid::ZeroDriftMode& mode);
    void GetZeroDriftMode(nn::hid::ZeroDriftMode& mode) const;
}
```

To enable zero-point drift correction, call `EnableZeroDrift`; to disable it, call `DisableZeroDrift`. Correction is enabled by default. Call `IsEnableZeroDrift` to get the current setting; a return value of `true` indicates that drift correction is enabled. When enabled, the previously set rotation matrix is applied to the output value. The return value from a call to `GetZeroDriftEffect` indicates the degree of drift correction being applied. A value of 0 indicates that no correction has been performed (movement was detected). Values approaching 1.0 indicate stability (no movement).

The degree of correction and the angular velocity at which the system is considered at rest both depend on the drift correction mode. Set and get the zero-point drift correction mode using the `SetZeroDriftMode()` and `GetZeroDriftMode()` functions, and use `ResetZeroDriftMode` to revert to the initial value (`GYROSCOPE_ZERODRIFT_STANDARD`).

**Table 6-3. Gyro Sensor Zero-Point Drift Correction Mode**

| Definition | Description |
|---|---|
| `GYROSCOPE_ZERODRIFT_LOOSE` | Correction is applied more loosely, and constant-velocity movement might not be detected. |
| `GYROSCOPE_ZERODRIFT_STANDARD` | Standard correction. (Default.) |
| `GYROSCOPE_ZERODRIFT_TIGHT` | Correction is applied strictly, allowing the detection of more precise movements. |

The zero-point offset may vary by up to 50 dps when zero-point drift correction is disabled. An application must adequately account for this variance when disabling correction, such as when detecting extremely small movements.

**Note:** Unless `GYROSCOPE_ZERODRIFT_LOOSE` is specified, the correction may not work as well even when the system is at rest. We recommend implementing your application to account for this, such as by applying `GYROSCOPE_ZERODRIFT_LOOSE` correction for scenes where the user is not expected to move the system, and then returning to the previously used correction mode during other use.

Set the zero-point play correction to avoid reacting to small changes in angular velocity. The play correction setting is applied equally to all directions.

**Code 6-6. Gyro Sensor Zero-Point Play Correction**

```
class nn::hid::GyroscopeReader
{
    void EnableZeroPlay();
    void DisableZeroPlay();
    bool IsEnableZeroPlay() const;
    f32 GetZeroPlayEffect() const;
    void SetZeroPlayParam(f32 radius);
    void GetZeroPlayParam(f32& radius) const;
```

```
    void ResetZeroPlayParam();
}
```

Call the `EnableZeroPlay()` function to enable zero-point play correction and call `DisableZeroPlay` to disable it. Correction is disabled by default. Call `IsEnableZeroPlay` to get the current setting; a return value of `true` indicates that play correction is enabled. The return value from a call to `GetZeroPlayEffect` indicates the degree of play correction being applied. The return value is negative when correction is disabled and 0 or greater when correction is enabled. As the angular velocity approaches the currently set play value, the return value approaches 0. A return value of exactly 0 indicates no correction.

To set the absolute value of the angular velocity for which play correction is 0, call the `SetZeroPlayParam()` function; to get the current setting, call `GetZeroPlayParam`. Call `ResetZeroPlayParam` to revert to the initial value of 0.005. An absolute angular velocity of 360 dps is represented by a value of 1.0.

If both zero-point drift and play correction are enabled, values are corrected first for drift and then for play.

Use the `nn::hid::GyroscopeReader` class to apply axial rotation to angular velocity after correcting for zero-point drift and play. The class calculates the rotation matrix based on output values when the system is at rest, allowing you to implement gyro sensor calibration in your application. You can also use axial rotation of output values to apply any angle tilt to the gyro sensor.

**Code 6-7. Gyro Sensor Axial Rotation**

```
class nn::hid::GyroscopeReader
{
    void EnableAxisRotation();
    void DisableAxisRotation();
    bool IsEnableAxisRotation() const;
    void SetAxisRotationMatrix(const nn::math::MTX34 mtx);
    void GetAxisRotationMatrix(nn::math::MTX34* pMtx) const;
    void ResetAxisRotationMatrix();
}
```

To enable axial rotation of angular velocity, call `EnableAxisRotation`; to disable, call `DisableAxisRotation`. Rotation is disabled by default. Call `IsEnableAxisRotation` to get the current setting; a return value of `true` indicates that rotation is enabled. When enabled, the currently set rotation matrix is applied to the angular velocity.

To set the rotation matrix (a 3 × 4 matrix), call `SetAxisRotationMatrix`; to get the currently set matrix, call `GetAxisRotationMatrix`. Call `ResetAxisRotationMatrix` to revert to the identity matrix (no rotation).

The `nn::hid::GyroscopeReader` class uses the accelerometer input value to correct the gyro sensor's 3D attitude. Anytime you use the gyro sensor, you are also using the accelerometer. When passing NULL as the value of the *`pAccelerometerReader`* parameter to the (overloaded) constructor, the `GyroscopeReader` class internally creates and uses an instance of the `nn::hid::AccelerometerReader` class that has the default settings. Pass a pointer to a configured instance of the `nn::hid::AccelerometerReader` class if you want to use different values for input play and detection sensitivity.

**Code 6-8. 3D Attitude Correction Using the Accelerometer**

```
class nn::hid::GyroscopeReader
{
    GyroscopeReader(nn::hid::AccelerometerReader* pAccelerometerReader = NULL,
                    nn::hid::Gyroscope& gyroscope = nn::hid::GetGyroscope());
```

```
    void EnableAccRevise();
    void DisableAccRevise();
    bool IsEnableAccRevise() const;
    f32 GetAccReviseEffect() const;
    void SetAccReviseParam(f32 revise_pw, f32 revise_range);
    void GetAccReviseParam(f32& revise_pw, f32& revise_range) const;
    void ResetAccReviseParam();
}
```

To enable correction using the accelerometer, call the `EnableAccRevise()` function; to disable, call `DisableAccRevise`. Call `IsEnableAccRevise` to get the current setting; a return value of `true` indicates that correction is enabled. The return value from a call to `GetAccReviseEffect` indicates the degree of correction being applied. The return value is always 0 when correction is disabled and 0 or greater when correction is enabled. The return value approaches 0 as the gyroscope's 3D attitude direction (`GyroscopeStatus.direction`) approaches the accelerometer's direction.

To set the accelerometer correction parameters (weight and enabled range), call `SetAccReviseParam`; to get the current settings, call `GetAccReviseParam`. Call `ResetAccReviseParam` to revert to the initial settings of 0.03 for weight and 0.4 for enabled range. For the *`revise_pw`* parameter, specify a value of `0.0` through `1.0` for the accelerometer weight. Higher values mean more severe correction. For the *`revise_range`* parameter, specify the accelerometer range to correct within. The range is `1.0` to the ±*`revise_range`* value. For example, specify a value of `0.4` to apply correction to acceleration values from 0.6 G through 1.4 G. The correction parameters are applied equally to all directions.

When applying angular velocity axial rotation to the gyro sensor with accelerometer correction enabled, specify the same rotation matrix for use with both the gyro sensor and accelerometer. When using an instance created with the default settings, the library makes sure internally that the same matrix is used, but if you use an instance created with any other settings, you must do this yourself. If different matrices are used, the accelerometer correction might not work properly.

## 6.2.5. Debug Pad (Control Pad for Debugging)

Call the `nn::hid::DebugPadReader` class's `Read` or `ReadLatest()` functions to get input from the digital buttons and two analog sticks on the debug pad as an `nn::hid::DebugPadStatus` structure. The `Read()` function can get sampling results in order of the latest result, but it cannot reacquire sampling results. Sampling results cannot be obtained if the function is called more frequently than the sampling frequency of 16 ms. On the other hand, the `ReadLatest()` function can get only the latest sampling result. It can also get sampling results again, so even when it is called at a cycle faster than the sampling cycle, it can still get sampling results.

The `nn::hid::DebugPadStatus` structure's `hold` member records which button was held at the time of sampling, the `trigger` member records which button was pressed at the time of sampling, and the `release` member records which button was released at the time of sampling. For the `ReadLatest()` function, both the `trigger` and `release` members are evaluated for their states at that time, so any changes between calls are not necessarily applied. The `leftStickX` and `leftStickY` members record the left analog stick's x and y axes, and the `rightStickX` and `rightStickY` members record the right analog stick's x and y axes as values ranging from -1.0 to 1.0.

Call the `SetStickClampMode()` function to set the analog stick clamp mode, and call `GetStickClampMode()` to get the current setting. This value cannot be set independently for the left and right analog sticks. You can choose either the `STICK_CLAMP_MODE_CIRCLE_WITH_PLAY` or `STICK_CLAMP_MODE_CIRCLE_WITHOUT_PLAY` clamp mode.

**Table 6-4. Debug Pad Digital Buttons and Definitions**

| Definition | Corresponding Button |
|---|---|
| DEBUG_PAD_BUTTON_UP | +Control Pad (Up) |
| DEBUG_PAD_BUTTON_DOWN | +Control Pad (Down) |
| DEBUG_PAD_BUTTON_LEFT | +Control Pad (Left) |
| DEBUG_PAD_BUTTON_RIGHT | +Control Pad (Right) |
| DEBUG_PAD_BUTTON_A | A Button |
| DEBUG_PAD_BUTTON_B | B Button |
| DEBUG_PAD_BUTTON_X | X Button |
| DEBUG_PAD_BUTTON_Y | Y Button |
| DEBUG_PAD_TRIGGER_L | L Trigger |
| DEBUG_PAD_TRIGGER_R | R Trigger |
| DEBUG_PAD_TRIGGER_ZL | ZL Trigger |
| DEBUG_PAD_TRIGGER_ZR | ZR Trigger |
| DEBUG_PAD_BUTTON_PLUS | Plus (+) Button |
| DEBUG_PAD_BUTTON_MINUS | Minus (-) Button |
| DEBUG_PAD_BUTTON_HOME | HOME Button |

## 6.2.6. Circle Pad Pro

The Circle Pad Pro is an optional peripheral device that is attached to the CTR for use. Using the Circle Pad Pro in addition to the standard input devices provided with the CTR system allows you to make use of the circle pad installed in the Circle Pad Pro (hereafter called the *Right Circle Pad*).

The circle pad installed by default in the CTR system is simply called the Circle Pad, and the digital buttons (the ZL and ZR Buttons). Input from input devices installed in the Circle Pad Pro is transmitted to the CTR system using infrared communication. The sampling cycle covers a range from 8 ms to 32 ms in 1-ms intervals. For a detailed description of how to perform settings, see 6.2.6.4. Sampling Start and State Acquisition.

**Note:** For the process flow for using Circle Pad Pro in applications, see Appendix: Process Flows for Using the Circle Pad Pro.

For more information about the C Stick, ZL Button, and ZR Button on SNAKE and for the differences from the Circle Pad Pro, see 6.2.7. C Stick.

**Note:** CTR uses infrared to communicate with the Circle Pad Pro.
If you want to use the Circle Pad Pro from an application while another feature is using the infrared communication, you must end the other feature first.

Infrared communication is used by the following features.

- Infrared communication between systems
- NFP (Only CTR)

## 6.2.6.1. Hardware Internal States

The Circle Pad Pro hardware has only two states: the *Active State* in which communication with the CTR system is possible, and the *Standby State* in which no communication is performed.

**Table 6-5. Hardware Internal States**

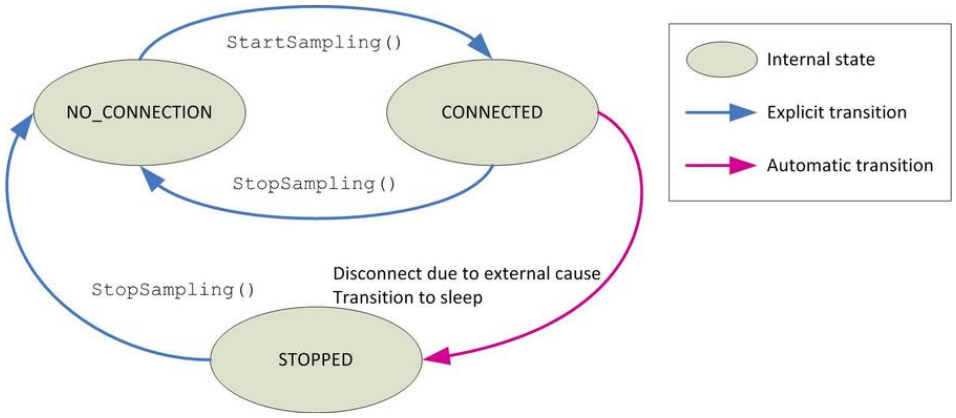
| Internal State | Description |
|---|---|
| Active State | State in which communication with the CTR is possible. Connection to the CTR can be made only in this state.<br>The device enters this state as soon as a battery is inserted. |
| Standby State | No communication with the CTR is performed in this state. When not in use, battery consumption is kept to a minimum to extend battery life.<br>The device enters this state if button input and infrared communication are not performed for a period of five minutes. The device returns to Active State by pushing one of the digital buttons on the Circle Pad Pro (ZL, ZR, or R). |

**Note:** The Circle Pad Pro has no indicator to display its internal state. The device's internal state cannot be obtained from the library.

## 6.2.6.2. Software Internal State

Software (library) internal states transition as shown in the following diagram.

**Figure 6-9. Software Internal State Transition (Circle Pad Pro)**




Connection status of internal states with the Circle Pad Pro is summarized in the following table.

**Table 6-6. Software Internal States**

| Internal State | Description |
|---|---|
| NO_CONNECTION | Connection with the Circle Pad Pro has not been established in this state.<br>This is the same internal state as immediately after initialization. |
| CONNECTED | Connection with the Circle Pad Pro is established and sampling is being performed in this state. |
| | Connection with the Circle Pad Pro, which had been established up to this point, has been interrupted due to an external cause (such as battery |

| | |
|---|---|
| STOPPED | consumption or detachment from the CTR system).<br>This internal state is also entered when the CTR system transitions to Sleep Mode. |

This internal state is defined by the `nn::hid::ExtraPad::ConnectionState` enumerator, taking a `CONNECTION_STATE_*` value.

## 6.2.6.3. Initialization

Before Circle Pad Pro input sampling can begin, the `nn::hid::ExtraPad` class initialization function must be called.

**Code 6-9. Circle Pad Pro Initialization Function**

```
static void nn::hid::ExtraPad::Initialize(
                void* workingMemory,
                size_t workingMemorySize);
```

Specify the buffer for infrared communication in *workingMemory*. The specified buffer must be of size `nn::hid::CTR::ExtraPad::WORKING_MEMORY_SIZE` (12,288 bytes), and the starting address alignment must be `nn::hid::CTR::ExtraPad::WORKING_MEMORY_ALIGNMENT` (4096 bytes). **Buffers allocated from device memory cannot be used.**

After initialization completes, the `nn::hid::ExtraPadReader` class can be used.

## 6.2.6.4. Sampling Start and State Acquisition

Use the following functions to begin sampling and get the state.

**Code 6-10. Functions Used to Start Sampling and Get States**

```
class nn::hid::ExtraPad
{
    static nn::Result StartSampling(s32 samplingThreadPriority, s32 period);
    static ConnectionState GetConnectionState();
    static bool IsSampling();
}
```

Circle Pad Pro sampling is performed by calling the `nn::hid::ExtraPad::StartSampling()` function. Calling the function before initialization has completed generates an error with the same value as the `nn::hid::MakeResultNotInitialized()` function. Processing of this function requires 50 to 200 ms when connection with the Circle Pad Pro has succeeded, and 100 ms when connection has failed.

Specify a sampling thread priority level in *samplingThreadPriority*. The specifiable range is 0 to 31. The higher the priority (the closer to 0), the more stable sampling becomes.

Specify the sampling cycle (in milliseconds) in *period*. The specifiable range is 8 to 32.

When sampling is started, application resources are consumed and the sampling thread is created. (The thread priority level is the value specified in *samplingThreadPriority*). This thread receives sampling data from the Circle Pad Pro according to the cycle setting specified in *period*, and sends approximately once every second for continuous communication with the

Circle Pad Pro. The higher the value specified for the sampling cycle, the larger the processing volume per unit of time. The processing burden on the application core and system core grows in proportion to the length of the cycle. To lighten the load on the system, it is most effective to set a lower value for the sampling cycle.

**Warning:** When the load from other processes is especially high, it interferes with sampling thread operating cycles and may cause delays in input, or disconnection of the Circle Pad Pro.

Sampling cannot start unless the Circle Pad Pro is in the Active State. It is not possible to determine whether the device is in the Active State from the library. Prompt the user to enter input from one of the digital buttons (R/ZR/ZL) on the Circle Pad Pro when the `StartSampling()` function sends back a return value indicating that the Circle Pad Pro cannot be found (the same value as for the `nn::hid::MakeResultNoConnection()` function). After you have returned to the Active State, you can reconnect.

When the `StartSampling()` function is called, a connection is made after a disconnect has been performed once.

**Note:** To perform ongoing detection of the Circle Pad Pro, a load must be set at less than a 32-ms cycle, assuming the `StartSampling()` function is called at a frequency of once per second. This is smaller than sampling processing.

The `nn::hid::ExtraPad::GetConnectionState()` function returns the Circle Pad Pro connection state as an internal state. Reconnect after performing a disconnect once, when a disconnect occurs due to an external cause and `CONNECTION_STATE_STOPPED` is returned.

The `nn::hid::ExtraPad::IsSampling()` function returns whether sampling is in progress. If the value `true` is returned, sampling is being performed and Circle Pad Pro input is applied to the `nn::hid::ExtraPadStatus` structure obtained by the `nn::hid::ExtraPadReader` class. When the value `true` is returned, it guarantees that the internal state is also in a connected state (CONNECTED), but note that if a connection or disconnect is underway, the value `false` could be returned even if the internal state is in a connected state.

**Note:** After Circle Pad Pro sampling begins, the `nn::hid::PadReader` class can no longer be used, but even if the Circle Pad Pro is not connected, input from the system's digital buttons and circle pad is applied to the `nn::hid::ExtraPadStatus` structure obtained by the `nn::hid::ExtraPadReader` class.

It is unnecessary for the application supporting the Circle Pad Pro to use the connection state to switch between classes and structures, because it uses the `nn::hid::ExtraPadReader` class and the `nn::hid::ExtraPadStatus` structure.

## 6.2.6.5. Stop Sampling

Call the `nn::hid::ExtraPad::StopSampling()` function to stop sampling.

**Code 6-11. Function Used to Stop Sampling**

```
static nn::Result nn::hid::ExtraPad::StopSampling();
```

After a disconnect has succeeded, the sampling thread is destroyed and the internal state transitions to a not connected state (NO_CONNECTION). This function always succeeds in processing as long as initialization has already been completed when it is called.

## 6.2.6.6. Getting Sampling Results

You can get sampling results with the Read or ReadLatest() function under the nn::hid::ExtraPadReader class, reflecting the nn:: hid::ExtraPadStatus structure. The Read() function can get sampling results in order of the latest result, but it cannot reacquire sampling results. When the function is called at a cycle faster than the sampling cycle, it cannot get sampling results. On the other hand, the ReadLatest() function can get only the latest sampling result. It can also get sampling results again, so even when it is called at a cycle faster than the sampling cycle, it can still get sampling results.

**Code 6-12. nn::hid::ExtraPadStatus Structure**

```
struct nn::hid::ExtraPadStatus {
    AnalogStickStatus stick;
    AnalogStickStatus extraStick;
    bit32 hold;
    bit32 trigger;
    bit32 release;
    u8    batteryLevel;
    bool  isConnected;
    NN_PADDING2;
};
```

The Circle Pad Pro's input from the system is applied to stick, in the same way as in the nn::hid::PadStatus structure.

Input from the Circle Pad Pro's Right Circle Pad is applied to extraStick.

The hold, trigger, and release member variables are the same as those in the nn::hid::PadStatus structure. The following buttons have been added to the device.

**Table 6-7. Buttons Added to the Extra Pad Reader**

| Definition | Corresponding Button |
|---|---|
| BUTTON_ZL | Circle Pad Pro ZL button |
| BUTTON_ZR | Circle Pad Pro ZR button |
| BUTTON_EMULATION_R_UP | Up on the Right Circle Pad emulates up on the +Control Pad |
| BUTTON_EMULATION_R_DOWN | Down on the Right Circle Pad emulates down on the +Control Pad |
| BUTTON_EMULATION_R_LEFT | Left on the Right Circle Pad emulates left on the +Control Pad |
| BUTTON_EMULATION_R_RIGHT | Right on the Right Circle Pad emulates right on the +Control Pad |

**Note:** When the Circle Pad Pro is attached, it is difficult to press the R Button, so the Circle Pad Pro has been fitted with its own R Button. The application cannot detect whether the pressed R Button is the one on the CTR system or on the Circle Pad Pro.

When the Circle Pad Pro is not performing sampling, input from the CTR system is applied to the nn::hid::ExtraPadStatus structure in 4-ms sampling cycles by the

`nn::hid::ExtraPadReader` class. However, when Circle Pad Pro sampling is being performed, the sampling cycle for input from the CTR system uses the sampling cycle specified by the `nn::hid::ExtraPad::StartSampling()` function (between 8 and 32 ms).

Circle Pad Pro battery level is stored in two values (0 and 1) in *`batteryLevel`*. Even when the value 0 is returned, there is approximately one day remaining of continuous operation.

**Warning:** Even if the value 1 is returned for remaining battery level, inserting a battery whose battery life is already partially consumed may result in the Circle Pad Pro failing to restart. If communication no longer works after inserting the battery, replace it with a new battery. This phenomenon occurs only when inserting a battery. After a new battery has been inserted, normal operation continues until the remaining battery life reaches 0.

Results of library searches for whether sampling is in progress are stored in *`isConnected`*.

**Note:** The time required until the application can get Circle Pad Pro input (input delay) is approximately 2 ms or more (2 ms + sampling cycle) when the volume of other processing being performed by the application is sufficiently small. It takes 0 sampling cycles or more to get user input from the Circle Pad Pro, plus communication time and CTR processing, which requires approximately 2 ms.

## 6.2.6.7. Circle Pad Clamp Processing

Functions used by the CTR system for circle pad clamp processing are defined with the same arguments as `nn::hid::PadReader` class member functions. Operation is also the same as with this class.

Functions used in Right Circle Pad clamp processing use the same settings as the circle pad on the CTR system side, except that the function name `Stick` is used instead of `ExtraStick`. These settings can be performed individually.

**Code 6-13. Functions Used in Right Circle Pad Clamp Processing**

```
class nn::hid::ExtraPadReader
{
    void SetExtraStickClamp(s16 min, s16 max);
    void GetExtraStickClamp(s16* pMin, s16* pMax) const;
    StickClampMode GetExtraStickClampMode() const;
    void SetExtraStickClampMode(StickClampMode mode);
    f32 NormalizeExtraStick(s16 x);
    void NormalizeExtraStickWithScale(
            f32* normalized_x, f32* normalized_y, s16 x, s16 y);
    void SetNormalizeExtraStickScaleSettings(f32 scale, s16 threshold);
    void GetNormalizeExtraStickScaleSettings(f32* scale, s16* threshold) const;
}
```

## 6.2.6.8. Event Notification

You can register events (`nn::os::LightEvent` class) that notify changes in the connection state and completion of sampling.

**Code 6-14. Functions Used in Event Notification**

```
class nn::hid::ExtraPad
{
    static void RegisterConnectionEvent(nn::os::LightEvent* pLightEvent);
    static void UnregisterConnectionEvent();
    static void RegisterSamplingEvent(nn::os::LightEvent* pLightEvent);
    static void UnregisterSamplingEvent();
}
```

Event notifications associated with changes in connection state are registered and unregistered with the `*ConnectionEvent()` functions, and event notifications for completing sampling are registered and unregistered with the `*SamplingEvent()` functions. After an event is registered, thread processing for sampling increases just slightly.

The processing load during registration of events that notify of changes in the connection state is so minor that it can be ignored. However, registration of the sampling complete event requires processing of each sampling cycle, so there is more processing load on the CPU. We recommend that these functions only be used on applications where event notification is used.

Changes in connection state can also be determined by `nn::hid::ExtraPad` class member functions, `GetConnectionState()` and `IsSampling()`, and by the `isConnected` structure member variable of `nn::hid::ExtraPadStatus`.

## 6.2.6.9. Calibration Applet

A library applet called the *Circle Pad Pro calibration applet* is provided to calibrate the feel of controls on the Right Circle Pad within the Circle Pad Pro. Follow the guidelines that provide examples where applications supporting the Circle Pad Pro can call up the Circle Pad Pro calibration applet.

For more information, see 12.1.7. Circle Pad Pro Calibration Applet.

## 6.2.6.10. Differences Between the Circle Pad and the Right Circle Pad

Input coordinates sampled from the Circle Pad and the Right Circle Pad have a tendency to change as a result of electrical fluctuations, even when operations are performed intentionally using the pad directly. The amount of change in values when key positions on the pads are fixed and the probability that this might occur differs for the Circle Pad and the Right Circle Pad, as indicated in the following table. The Right Circle Pad is especially susceptible to these changes, so caution is required.

**Table 6-8. Probability That Circle Pad and Right Circle Pad Input Coordinates Will Change**

| Amount of Change | Circle Pad x-coordinate | Circle Pad y-coordinate | Right Circle Pad x-coordinate | Right Circle Pad y-coordinate |
|---|---|---|---|---|
| -3 | 0.000006 % | 0.000006 % | 0.002 % | 0.001 % |
| -2 | 0.000178 % | 0.000178 % | 0.349 % | 0.352 % |
| –1 | 0.003541 % | 0.003541 % | 3.485 % | 3.273 % |

| | | | | |
|---|---|---|---|---|
| ±0 | 99.949837 % | 99.949837 % | 92.326 % | 92.743 % |
| +1 | 0.003541 % | 0.003541 % | 3.491 % | 3.285 % |
| +2 | 0.000178 % | 0.000178 % | 0.346 % | 0.345 % |
| +3 | 0.000006 % | 0.000006 % | 0.002 % | 0.001 % |

The probability of change occurring within the unit of time noted is determined according to the following formula:

Amount of sampling data obtained during unit of time × Probability from Table 6-8.

.

If the pad is positioned in the center after the user's finger is removed, the value will not change from the 0 position. This is because even with some variation in value, the minimum clamping value is never exceeded.

## 6.2.7. C Stick

The library allows you to use the C Stick on SNAKE without differentiating from the Right Circle Pad on the Circle Pad Pro, but the actual hardware is different. You must consider hardware differences when using this approach.

### 6.2.7.1. Properties of C Stick Hardware

The C Stick on SNAKE is an analog input device that detects minor deformations (strain) when pressure is applied to a resin stick and reads the magnitude of the deformation as changes in voltage.

Note the following points because the output value from the C Stick depends on the pressure on the stick part.

- It may be difficult to maintain the maximum input value depending on the strength of the user.
- When too much pressure is applied, the output value from the C Stick does not change (the output value is saturated).
- When the C Stick is strongly pressed from directly above, the output value from the C Stick is undefined.
- When the X Button is strongly held down, the C Stick output value might fluctuate due to the close physical proximity to the X Button.

In addition, due to variances in the performance of the sensors on each system, note the following.

- Sensitivity variances between individual systems have an effect on resolution.

#### Effect on the Pressure Required to Input the Maximum Value

The maximum detectable pressure range of the C Stick is approximately 240 gf. Consequently, it might be difficult for some users to apply the maximum pressure continuously.

The minimum pressure value differs depending on the clamping method and input direction. The detectable pressure range of the C Stick is summarized in the following table.

**Table 6-9. Detectable Pressure Range of C Stick**

| Clamping Method | Minimum Detectable Pressure | | Maximum Detectable Pressure |
|---|---|---|---|
| | Up, Down, Left, Right (4 directions) | Diagonal 45° directions | |
| Circular (min=40, max=145) | Approx. 68 gf | Approx. 72 gf | Approx. 240 gf |
| Cross (min=36, max=145) | Approx. 63 gf | Approx. 90 gf | |
| Minimal (min=40, max=145) | Approx. 63 gf | Approx. 72 gf | |

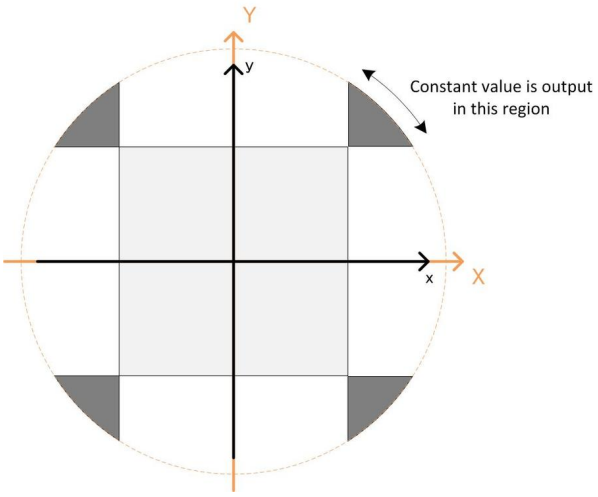
## Effect of Saturated Output Value

When excessive force is applied to the C Stick, the output value of the C Stick no longer changes, even if more pressure is applied. This condition is called "saturation of the output value."

C Stick input is measured on two independent axes (the x-axis and y-axis). In a situation where the output value is saturated for both axes, the system can no longer detect any C Stick operations by the user.

For example, as illustrated in the following figure, if the C Stick is moved in a large circular motion with enough force and the C Stick input axes are aligned with the system, the output value does not change for some regions at ±45° angles from the input axes.

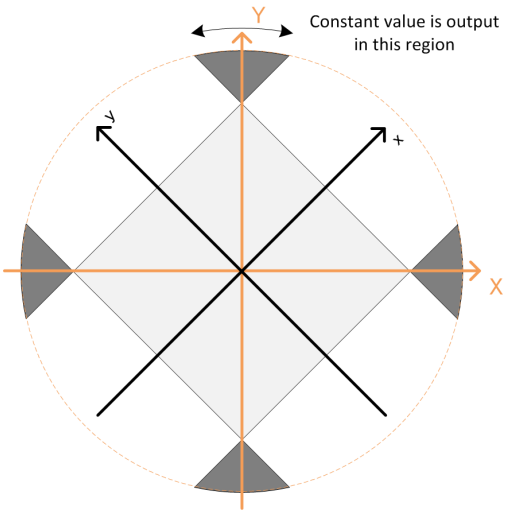
**Figure 6-10. Saturation of Output Values (C Stick Input Axes Aligned With System)**




The orange x-axis and y-axis show the orientation of the system, and the black x-axis and y-axis show the input axis of the C Stick. The orange circle shows the input pressure, the light shaded area (in the middle) shows where a raw output value can be obtained from the C Stick device, and the dark shaded area shows where the output value does not change for either axis. The area where the value does not change is drawn larger than the actual area for illustrative purposes.

The C Stick input axes are rotated 45° from the orientation of the system to minimize problems from the user's perspective.

**Figure 6-11. Saturation of Output Values (C Stick Input Axes Rotated 45° From System Orientation)**




The rotation of the input axes is handled by the system. Applications do not need to apply any rotation to C Stick input retrieved from the library.

### Behavior When Strongly Pressed From Above

This action causes the stick part to bend, which may result in the following symptoms.

- The C Stick output value fluctuates even though no directional force is applied.
- The input is processed as a different direction than what the user intended.

### Effect of Holding Down the X Button

When strong pressure is applied to the X Button, the C Stick output value might fluctuate even if it is not being used. This issue occurs due to the deformation of the housing when the X Button is strongly pressed, which can also cause the C Stick to bend.

This issue does not occur when the X Button is rapidly pressed, and requires significant force to occur when holding down the X Button. More force is required to reproduce this issue on the development hardware and the new Nintendo 3DS than on the new Nintendo 3DS XL.

In practice, Nintendo expects the effect of this issue on the C Stick output value to stay within the following ranges.

**Table 6-10. C Stick Output Value Fluctuation Range Caused by Holding X Button**

| Clamping Method | x-axis | y-axis |
|---|---|---|
| Circular (min=40, max=145) | 0 to 10 | 0 to -35 |
| Cross (min=36, max=145) | 0 | 0 to –38 |
| Minimal (min=40, max=145) | 0 to 10 | 0 to -37 |

### Effect of Sensitivity Variations Between Individual Systems on Resolution

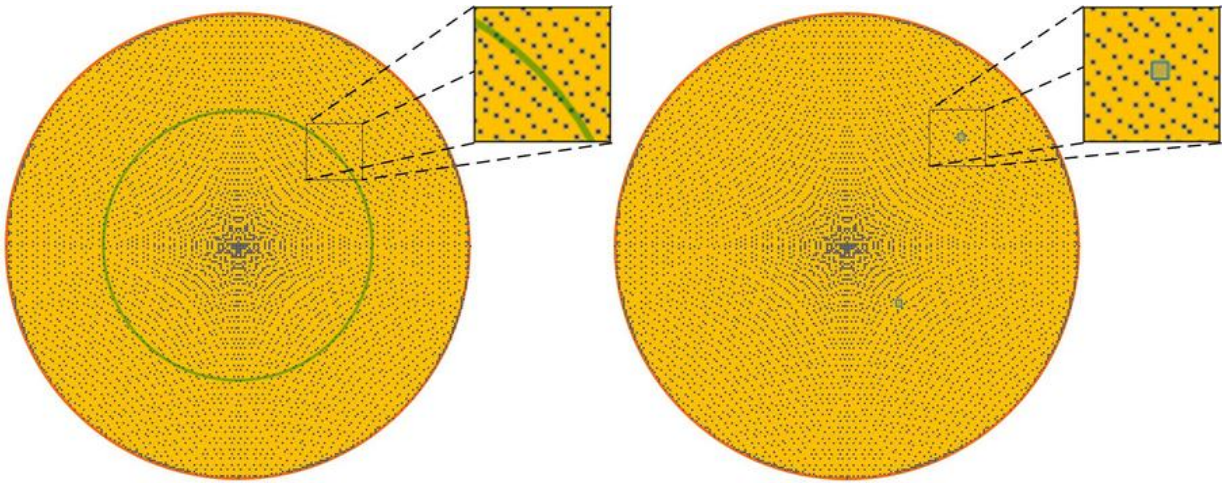
The C Stick has slight differences in sensitivity between individual systems and is calibrated during the manufacturing process to mitigate these differences. Due to the calibration process, some systems have higher resolution than others.

Depending on the individual system and the clamping mode, it may not be possible to get all of the values within the clamping range from the C Stick. The change in value is not guaranteed to be consecutive on systems with higher resolution, and this tendency becomes even more pronounced on systems with lower resolution. Output is guaranteed, however, for the center point and the outer edge.

For this reason, Nintendo does not recommend implementations that require input from a narrow range or expect a smooth change in value.

The following figure shows the output value distribution on a low-resolution system with circular clamping applied, in addition to an example of the narrow detection range that is not recommended. The orange background represents the range of values available to the application, and the black points represent the actual values that are output. Note that the values of adjacent black dots are not guaranteed to be consecutive. The green circle on the left side shows a narrow detection range when the C Stick output value is converted to a vector length. The green square on the right shows a narrow detection range when using the raw C Stick output value. In other words, depending on the input direction, there might not be any points that satisfy the required vector length, and there might not be any points at the raw value being detected.

**Figure 6-12. Distribution of C Stick Output Values and Examples of Narrow Detection Ranges (Circular Clamping, Low Resolution System)**



## 6.2.7.2. Differences From the Circle Pad Pro

When using input to the SNAKE C Stick with the HID library, note the following differences from the specifications and behavior of the Circle Pad Pro.

- The configurable sampling frequency is different.
- The device is never disconnected due to external factors.
- The behavior when waking from Sleep Mode is different.
- The center point and sensitivity are not calibrated in the Circle Pad Pro Calibration Applet.

### Different Configurable Sampling Frequency

A sampling frequency of 10 to 21 ms can be set for the SNAKE C Stick, ZL Button, and ZR Button hardware.

The `nn::hid::ExtraPad::StartSampling()` function allows you to specify a frequency in the range from 8 to 32 ms, but the hardware handles values below 10 ms as 10 ms, and values above 21 ms as 21 ms. The function returns data at that frequency.

We recommend setting a sampling frequency of 10 to 21 ms based on the sampling frequency of the hardware.

### No Disconnects due to External Factors

The Circle Pad Pro, which was connected using infrared communication, could be disconnected by external factors such as being detached or a dead battery. The SNAKE C Stick, ZL Button, and ZR Button, however, are part of the system itself, so `nn::hid::ExtraPad::GetConnectionState()` never returns `nn::hid::ExtraPad::CONNECTION_STATE_STOPPED` due to external factors.

Because the battery never runs out, the `batteryLevel` member of the `nn::hid::ExtraPadStatus` structure is always set to 1 while input is being sampled.

### Different Behavior When Waking From Sleep Mode

The Circle Pad Pro and the C Stick have the following differences in behavior when the system enters Sleep Mode without calling `nn::hid::ExtraPad::StopSampling`.

- Circle Pad Pro
  Infrared communication stops when the system enters Sleep Mode, which would cause sampling requests to fail immediately after the system wakes. To prevent this, the sampling group is removed and the library treats the Circle Pad Pro as having been stopped. However, because the internal state changes based on the infrared communication status, the internal state before detecting the stoppage is unstable.
  Specifically, the `nn::hid::ExtraPad::GetConnectionState()` function generally returns `CONNECTION_STATE_STOPPED`, but it might sometimes return `CONNECTION_STATE_CONNECTED`.
- C Stick
  Sampling requests succeed immediately after waking from Sleep Mode and the connection state is retained.
  Specifically, the `nn::hid::ExtraPad::GetConnectionState()` function always returns `CONNECTION_STATE_CONNECTED`.

The value returned by the `nn::hid::ExtraPad::IsSampling()` function also depends on the internal state, so that value changes at the same time.

To handle this behavior, we recommend calling `nn::hid::ExtraPad::StopSampling` to stop the sampling process before entering Sleep Mode. If you call `nn::hid::ExtraPad::StopSampling`, resume sampling by calling `nn::hid::ExtraPad::StartSampling` after waking from Sleep Mode. Nintendo also recommends this procedure before and after calling functions where the system could enter Sleep Mode before control returns to the application, such as the HOME Menu or a library applet.

### No Center Point or Sensitivity Calibration in Circle Pad Pro Calibration Applet

SNAKE works differently from CTR when the Circle Pad Pro Calibration Applet is called. On SNAKE, the system enters a mode to check the operation of the C Stick and displays a description of how the C Stick is calibrated.

This is because the C Stick center point is automatically calibrated at the following times.

- When power is turned on
  If the C Stick is pressed when the power is turned on, it will not be calibrated correctly. Even so, it is still possible to correctly calibrate the C Stick after waking from Sleep Mode, as described below.
- When waking from Sleep Mode
  It is physically impossible to operate the C Stick while the system is closed, so no C Stick input ever occurs during sleep. You can safely assume that the device was calibrated correctly in this case.
- During sampling (when the user is not using the control)
  The application does not need to consider the effects of this calibration because it takes place when the C Stick output value is below the lower limit of the clamping.

**Warning:** The center point calibration after Sleep Mode is triggered when the system wakes. Consequently, the calibration does not take place when the system is reopened if the application rejects the Sleep Mode request. The message that is displayed in the calibration applet tells the user to put the system to sleep by closing it.

Note that center point calibration does not take place when an application rejects the sleep request for one of the following reasons.

- The application does not want to disconnect even if the system is closed.
- The system does not enter Sleep Mode when closed while in Sound Mode if headphones are connected.

## 6.3. MIC Library

The MIC library handles audio input from the microphone obtained by automatic sampling.

Call the `nn::mic::Initialize()` function to initialize the MIC library. The microphone can be used after this function successfully completes, but to sample microphone input in an application, you must take care of other preparations required for sampling, such as allocating the buffers for storing the sampling results, configuring the microphone gain, and dealing with microphone power control.

### 6.3.1. Buffer Allocation

The application allocates the buffer for storing sampling results. The starting address of the buffer must be `nn::mic::BUFFER_ALIGNMENT` (4096-byte) aligned, its size must be a multiple of `nn::mic::BUFFER_UNITSIZE` (4096 bytes), and **it must be allocated from non-device memory**. Pass the allocated buffer to the MIC library using the `nn::mic::SetBuffer()` function. The last 4 bytes of the buffer are set aside for management purposes, so the actual space available for storing sampling results is the value specified minus these 4 bytes. Use the `nn::mic::GetSamplingBufferSize()` function to get the size of the buffer used to store sampling results.

Calling the `nn::mic::SetBuffer()` function after a buffer is already allocated causes an error. To reallocate a buffer, first call the `nn::mic::ResetBuffer()` function and then call the `nn::mic::SetBuffer()` function.

## 6.3.2. Microphone Gain Settings

The microphone gain setting determines the amplification multiple for audio input from the microphone.

Use the `nn::mic::GetAmpGain()` and `nn::mic::SetAmpGain()` functions to get and set the gain. You can set the gain in the range from 0 to 119 with each increase of 1 representing a 0.5 dB increment. 0 equates to 10.5 dB (a multiple of roughly 3.4), and 119 equates to 70.0 dB (a multiple of roughly 3162). As shown in the following table, the four gain levels for the NITRO microphone can be used by converting to gain setting values.

**Table 6-11. NITRO Settable Multiples and Gain Settings**

| Ratio | dB | Gain Setting Values |
|---|---|---|
| 20 x | 26.0 | 31 |
| 40 x | 32.0 | 43 |
| 80 x | 38.0 | 55 |
| 160 x | 44.0 | 67 |

The microphone gain setting is set to `AMP_GAIN_DEFAULT_VALUE` when the library is initialized.

## 6.3.3. Microphone Power Control

Use the `nn::mic::SetAmp()` function to control power to the microphone (microphone amp). Pass `true` as an argument to turn the microphone on. Microphone input is unstable immediately after turning the microphone on or recovering from sleep, so the first second of sampling is forcibly muted.

Call the `nn::mic::GetAmp()` function to get the current setting.

## 6.3.4. Start Sampling

The preceding steps have prepared the microphone for sampling. Call the `nn::mic::StartSampling()` function to start sampling automatically.

**Code 6-15. Start Microphone Sampling**

```
nn::Result nn::mic::StartSampling(
        nn::mic::SamplingType type, nn::mic::SamplingRate rate, s32 offset,
        size_t size, bool loop);
```

Specify the type of data to get in the *type* parameter. You can choose from the following four types

of data, depending on factors such as your bit width and sign requirements.

**Table 6-12. Microphone Sampling Data Types**

| Type | Bit Width | Signed | Sampling Value Range | Sampling Value Indicating Mute |
|---|---|---|---|---|
| SAMPLING_TYPE_8BIT | 8 | None | 0 to 255 | 128 |
| SAMPLING_TYPE_16BIT | 16 | None | 0 to 65535 | 32768 |
| SAMPLING_TYPE_SIGNED_8BIT | 8 | Yes | –128 to 127 | 0 |
| SAMPLING_TYPE_SIGNED_16BIT | 16 | Yes | –32,768 to 32,767 | 0 |

Specify the sampling rate in the *rate* parameter. You can choose from the following four sampling rates.

**Table 6-13. Microphone Sampling Rates**

| Rate | Sampling Rate |
|---|---|
| SAMPLING_RATE_32730 | 32.73 kHz (32728.498046875 Hz) |
| SAMPLING_RATE_16360 | 16.36 kHz (16364.2490234375 Hz) |
| SAMPLING_RATE_10910 | 10.91 kHz (10,909.4993489583 Hz) |
| SAMPLING_RATE_8180 | 8.18 kHz (8182.12451171875 Hz) |

Specify the offset from the start of the sampling data storage location (the start of the buffer) in the *offset* parameter. An alignment greater than 0, and of `nn::mic::OUTPUT_OFFSET_ALIGNMENT` (2 bytes) must be specified.

Specify the number of bytes of sampling results to store as a multiple of `nn::mic::CTR::OUTPUT_UNITSIZE` (2) for each sampling result in the *size* parameter. This value must be small enough that (*offset* + *size*) is no larger than the size specified when allocating the buffer.

Specify whether to continue sampling after storing *size* bytes of sampling results in the *loop* parameter. Pass `true` to continue sampling and treat the buffer as a ring buffer (the region of *size* bytes starting from *offset*).

If you call the `nn::mic::StartSampling()` function while already sampling, the current sampling session is terminated and a new sampling session is started. If this function is called while the system is closed, it returns `nn::mic::ResultShellClose` and sampling does not begin. If the system is closed during ongoing sampling, sampling is automatically stopped and resumes when the system is opened.

Call the `nn::mic::AdjustSampling()` function to change the sampling rate during sampling. When you change the sampling rate this way, the new rate is used for sampling data starting from the current storage location.

Call the `nn::mic::IsSampling()` function to check whether the microphone is currently being sampled. However, this entails a significant processing burden because it sends a request directly to the device. It is not suitable for operations that involve calling at each frame.

Call the `nn::mic::SetLowPassFilter()` function and pass `true` in the *enable* parameter to apply a low-pass filter using a microphone input cutoff frequency of 45% of the sampling rate. The default value is `false` (no filtering). However, data sampled with a sampling rate of `SAMPLING_RATE_32730` was sampled using a low-pass filter, so applying a low-pass filter using this function has no effect.

### 6.3.4.1. Synchronization With Sound Processing

The DSP that processes sound and the processor that handles the microphone are separate devices. It is not possible to synchronize them perfectly. It is possible, however, to keep them in near synchronization by correcting differences in timing.

Keep the timers in sync by periodically calculating mismatches between the sampling count and elapsed time of the microphone, and the playback time of the audio, and correct any offset. Also correct mismatches in the sampling frequency by calling `Voice::SetPitch`.

## 6.3.5. Getting Sampling Results

Call the `nn::mic::GetLastSamplingAddress()` function to get the address of the most recent sampling results.

The `nn::os::Event` class instance, which is obtainable by a call to the `nn::mic::GetBufferFullEvent()` function, is a manually resetting event that enters the signaled state when the buffer cannot store any more sampling results. This event enters the signaled state when the buffer has stored *`size`* bytes of sampling results from the *`offset`* position and remains in the signaled state until the `ClearSignal()` function is called.

Due to differences in individual CTR systems in terms of the range of values obtained as input, the guaranteed input value ranges for each sampling type are defined by the `TYPE_*_GUARANTEED_INPUT_MIN(MAX)` constants. Do not design applications that expect values outside of these guaranteed input ranges when determining if there is microphone input.

**Table 6-14. Guaranteed Microphone Input Ranges for Each Sampling Type**

| Sampling Type | Lower Limit | Upper Limit |
|---|---|---|
| `SAMPLING_TYPE_8BIT` | 27 | 228 |
| `SAMPLING_TYPE_16BIT` | 7105 | 58415 |
| `SAMPLING_TYPE_SIGNED_8BIT` | -101 | 100 |
| `SAMPLING_TYPE_SIGNED_16BIT` | -25663 | 25647 |

The default behavior is for microphone input values to be clamped to within these guaranteed ranges. If you need input values across a broader range, you can disable clamping by calling the `nn::mic::SetClamp()` function, but note that you might still be unable to obtain input values from outside the guaranteed input ranges.

### 6.3.5.1. Prohibited Ranges for Microphone Input Determination

Call the `nn::mic::GetForbiddenArea()` function, passing in the microphone amplifier gain setting and sampling type, to get the lower and upper limits for sampling results that must not be recognized as microphone input. The application must not recognize any sampling results with values between these lower and upper limits as microphone input. The obtainable values are shown in Table 6-15.

The rows shaded in light gray in Table 6-15 show where the gain is 68 (44.5 dB) or higher. In these ranges, microphone input is subject to marked static and other noise due to sources such

as system speaker output, button clicks, and the sound of the stylus contacting the touch panel. These ranges are unsuitable for determining whether there is any microphone input based on amplitude levels. The rows in the table shaded in darker gray show where the gain is 104 (62.5 dB) or higher, where the full range of microphone input is noisy. Only use these ranges when invalid microphone input is not a problem.

**Table 6-15. Prohibited Ranges for Microphone Input Determination**

| Sampling Type | Gain | dB | Prohibited Ranges for Microphone Input Detection (Noise Component) |
|---|---|---|---|
| SAMPLING_TYPE_8BIT | 0 to 31 | 10.5 to 26.0 | 125 to 131 |
| | 32 to 43 | 26.5 to 32.0 | 123 to 133 |
| | 44 to 55 | 32.5 to 38.0 | 119 to 137 |
| | 56 to 67 | 38.5 to 44.0 | 112 to 144 |
| | 68 to 80 | 44.5 to 50.5 | 96 to 160 |
| | 81 to 91 | 51.0 to 56.0 | 77 to 179 |
| | 92 to 103 | 56.5 to 62.0 | 18 to 238 |
| | 104 to 119 | 62.5 to 70.0 | 0 to 255 |
| SAMPLING_TYPE_16BIT | 0 to 31 | 10.5 to 26.0 | 32000 to 33536 |
| | 32 to 43 | 26.5 to 32.0 | 31488 to 34048 |
| | 44 to 55 | 32.5 to 38.0 | 30464 to 35072 |
| | 56 to 67 | 38.5 to 44.0 | 28672 to 36864 |
| | 68 to 80 | 44.5 to 50.5 | 24576 to 40960 |
| | 81 to 91 | 51.0 to 56.0 | 19712 to 45824 |
| | 92 to 103 | 56.5 to 62.0 | 4608 to 60928 |
| | 104 to 119 | 62.5 to 70.0 | 0 to 65535 |
| SAMPLING_TYPE_SIGNED_8BIT | 0 to 31 | 10.5 to 26.0 | -3 to +3 |
| | 32 to 43 | 26.5 to 32.0 | -5 to +5 |
| | 44 to 55 | 32.5 to 38.0 | -9 to +9 |
| | 56 to 67 | 38.5 to 44.0 | -16 to +16 |
| | 68 to 80 | 44.5 to 50.5 | -32 to +32 |
| | 81 to | 51.0 to | |

| | | | |
|---|---|---|---|
| | 91 | 56.0 | -51 to +51 |
| | 92 to 103 | 56.5 to 62.0 | -110 to +110 |
| | 104 to 119 | 62.5 to 70.0 | -128 to +127 |
| SAMPLING_TYPE_SIGNED_16BIT | 0 to 31 | 10.5 to 26.0 | -768 to +768 |
| | 32 to 43 | 26.5 to 32.0 | -1280 to +1280 |
| | 44 to 55 | 32.5 to 38.0 | -2304 to +2304 |
| | 56 to 67 | 38.5 to 44.0 | -4096 to +4096 |
| | 68 to 80 | 44.5 to 50.5 | -8192 to +8192 |
| | 81 to 91 | 51.0 to 56.0 | -13056 to +13056 |
| | 92 to 103 | 56.5 to 62.0 | -28160 to +28160 |
| | 104 to 119 | 62.5 to 70.0 | -32768 to +32767 |

### 6.3.6. Stop Sampling

Use the `nn::mic::StopSampling()` function to stop sampling. This function only stops sampling. It does not turn off power to the microphone.

### 6.3.7. Closing the MIC Library

To stop using input from the microphone, such as when quitting an application, first stop sampling, and then do the following.

1. Call the `nn::mic::SetAmp()` function and pass `false` as an argument to turn off the microphone power.
2. Call the `nn::mic::ResetBuffer()` function to free up allocated buffers.
3. Call the `nn::mic::Finalize()` function to close the MIC library.

If you want to stop sampling for a time and then sample more later, just turn off the microphone power to save battery life.

The microphone gain setting is set to `AMP_GAIN_DEFAULT_VALUE` when the library is closed.

## 6.4. Camera Library and Y2R Library

The camera library handles operations for the camera on the system. Images captured from the

camera are only in YUV format. For conversion to RGB format, we recommend using the YUVtoRGB circuit, which also supports conversion to the native GPU format. Use the Y2R library for YUVtoRGB circuit operations.

## 6.4.1. Initializing

To initialize the camera library, call the `nn::camera::Initialize()` function; to initialize the Y2R library, call the `nn::y2r::Initialize()` function. However, calling the initializers is not sufficient to prepare the camera and the YUVtoRGB circuit. You must also configure the settings for each and ready them to send and receive data.

## 6.4.2. Photography Environment Settings

This section describes the configurable settings for the photography environment. You must specify which camera to configure by selecting from the following options.

**Table 6-16. Specifying Which Camera to Configure**

| Setting Value | Target Camera |
|---|---|
| `SELECT_NONE` | Specifies no cameras. Used to, for example, configure standby settings. |
| `SELECT_OUT1` | Right outer camera. |
| `SELECT_IN1` | Inner camera. |
| `SELECT_IN1_OUT1` | Right outer camera and inner camera. |
| `SELECT_OUT2` | Left outer camera. |
| `SELECT_OUT1_OUT2` | Both left and right outer cameras. |
| `SELECT_IN1_OUT2` | Right outer camera and inner camera. |
| `SELECT_ALL` | All cameras. Both left and right outer cameras and inner camera. |

Using both outer cameras together with `SELECT_OUT1_OUT2` is also called *stereo mode*, with the images taken by the outer cameras appearing in 3D. When doing so, make sure the photographic environment settings of the two cameras are the same.

**Note:** For more information about stereoscopic display and calibration methods using the stereo camera, see *3DS Programming Manual: Advanced Graphics*.

**Warning:** Configuring the photographic environment settings while capturing could cause distortion of the captured images. Consequently, first pause capture and then configure the settings.

When specifying `SELECT_ALL`, note that the cameras selected differ from in the past when there was only one outer camera.

Note that configuring the photographic environment settings while the camera is restarting may cause the library's processing to block for an extended period.

Image data taken by the camera includes variations based on the camera itself, ambient

light, and color variation in the subject. For example, camera variations include angle-of-view, resolution, color reproducibility, rotation, and distortion. An image taken with the same angle and camera setting is different in each system.

### Resolution

Call the `nn::camera::SetSize()` function to configure the resolution of the images captured by the camera. The resolution is the size of the image before trimming. Choose from the following resolution options.

**Table 6-17. Resolution**

| Setting Value | Resolution | Description |
|---|---|---|
| `SIZE_VGA` | 640×480 | VGA |
| `SIZE_QVGA` | 320×240 | QVGA |
| `SIZE_QQVGA` | 160×120 | QQVGA |
| `SIZE_CIF` | 352×288 | CIF (aspect ratio of 11: 9) |
| `SIZE_QCIF` | 176×144 | QCIF (aspect ratio of 11: 9) |
| `SIZE_DS_LCD` | 256×192 | Resolution of the DS LCD. |
| `SIZE_DS_LCDx4` | 512×384 | Resolution at twice the height and width of the DS LCD. |
| `SIZE_CTR_TOP_LCD` | 400×240 | Resolution of the 3DS upper LCD |
| `SIZE_CTR_BOTTOM_LCD` | 320×240 | Resolution of the 3DS lower LCD. Same as QVGA. |

Images captured using the `SIZE_CIF`, `SIZE_QCIF`, and `SIZE_CTR_TOP_LCD` settings have a 4:3 aspect ratio, with the left and right sides of the images trimmed.

Call the `nn::camera::SetDetailSize()` function to configure the other resolution settings. You can freely set the resolution of the captured images to crop from the original VGA-sized image by specifying the height and width of the output image. Make sure that the width and height are greater than or equal to the output image size. If not trimming, the output image width times the height must be a multiple of 128. To maintain compatibility, make sure that you specify the *`cropX0`* parameter as both an even number and a value that results in a multiple of 4 (*`cropX1`* - *`cropX0`* + 1).

To take images in stereo mode, make sure that the two cameras are set to the same resolution.

### Sharpness

Call the `nn::camera::SetSharpness()` function to set a camera's sharpness. Valid sharpness values range from -4 to +5.

### Exposure

Call the `nn::camera::SetExposure()` function to set the camera's exposure. Valid exposure values range from -5 to +5.

Call the `nn::camera::SetAutoExposure()` function to enable or disable the auto exposure (AE) feature. The exposure value is unstable immediately after starting up the camera, so we recommend leaving the AE feature enabled. Call the `nn::camera::IsAutoExposure()` function to get the camera's current AE setting. Note that the AE feature is enabled when setting the

exposure with the `nn::camera::SetExposure()` function, even if you have disabled the AE feature previously.

Call the `nn::camera::SetAutoExposureWindow()` function to set the region used as the standard for automatic calculation when in automatic exposure mode. Specify this region as a portion of the 640 × 480 VGA maximum size of a captured image, changing the starting coordinates, width, and height within the following ranges. The three rightmost columns of the table show the initial setting values for the inner, right outer, and left outer cameras.

**Table 6-18. Specifying the Standard Region for Exposure**

| Parameter | Setting | Setting Value Range | IN1 | OUT1 | OUT2 |
|---|---|---|---|---|---|
| *startX* | Starting coordinate (horizontal) | 0 to 600 (in 40-pixel increments) | 80 | 0 | 0 |
| *startY* | Starting coordinate (vertical) | 0 to 450 (in 30-pixel increments) | 60 | 0 | 0 |
| *width* | Width | 40 to 640 (in 40-pixel increments) 640 or less when summed with *startX* | 480 | 640 | 640 |
| *height* | Height | 30 to 480 (in 30-pixel increments) 480 or less when summed with *startY* | 360 | 480 | 480 |

### Frame Rate

Call the `nn::camera::SetFrame Rate()` function to set how many frames per second (fps) to capture. You can choose from the following values. When in stereo mode, use the same frame rate for both outer cameras.

**Table 6-19. Frame Rate**

| Setting Value | Frame Rate |
|---|---|
| `FRAME_RATE_15` | Fixed at 15 fps. |
| `FRAME_RATE_15_TO_5` | Automatically varies from 15 to 5 fps depending on the available light. |
| `FRAME_RATE_15_TO_2` | Automatically varies from 15 to 2 fps depending on the available light. |
| `FRAME_RATE_10` | Fixed at 10 fps. |
| `FRAME_RATE_8_5` | Fixed at 8.5 fps. |
| `FRAME_RATE_5` | Fixed at 5 fps. |
| `FRAME_RATE_20` | Fixed at 20 fps. |
| `FRAME_RATE_20_TO_5` | Automatically varies from 20 to 5 fps depending on the available light |
| `FRAME_RATE_30` | Fixed at 30 fps. |
| `FRAME_RATE_30_TO_5` | Automatically varies from 30 to 5 fps depending on the available light. |
| `FRAME_RATE_15_TO_10` | Automatically varies from 15 to 10 fps depending on the available light. |
| `FRAME_RATE_20_TO_10` | Automatically varies from 20 to 10 fps depending on the available light. |
| `FRAME_RATE_30_TO_10` | Automatically varies from 30 to 10 fps depending on the available light. |

### White Balance

Call the `nn::camera::SetWhiteBalance()` function to set the camera's white balance. Choose from the following white balance setting values.

**Table 6-20. White Balance**

| Setting Value | Aliases | Description |
|---|---|---|
| WHITE_BALANCE_AUTO | WHITE_BALANCE_NORMAL | Automatic white balance. |
| WHITE_BALANCE_3200K | WHITE_BALANCE_TUNGSTEN | Tungsten light (incandescent light bulb). |
| WHITE_BALANCE_4150K | WHITE_BALANCE_WHITE_FLUORESCENT_LIGHT | White fluorescent. |
| WHITE_BALANCE_5200K | WHITE_BALANCE_DAYLIGHT | Daylight. |
| WHITE_BALANCE_6000K | WHITE_BALANCE_CLOUDY | Cloud cover. |
| | WHITE_BALANCE_HORIZON | Sunset. |
| WHITE_BALANCE_7000K | WHITE_BALANCE_SHADE | Shade. |

Call the nn::camera::SetWhiteBalance() function and configure the white balance setting to WHITE_BALANCE_AUTO to enable the auto white balance feature. Any other setting value will disable the auto white balance feature.

Use the nn::camera::SetAutoWhiteBalance() function to enable or disable the auto white balance feature after enabling it by using WHITE_BALANCE_AUTO. If a white balance setting other than WHITE_BALANCE_AUTO was used, you cannot use the nn::camera::SetAutoWhiteBalance() function. Use the nn::camera::IsAutoWhiteBalance() function to get the current auto white balance setting.

Call nn::camera::SetAutoWhiteBalanceWindow to set the region used as the benchmark for auto white balance function when automatically calculating the white balance. Specify this region as a portion of the 640 × 480 VGA maximum size of a captured image, changing the starting coordinates, width, and height within the following ranges. The three rightmost columns of the table show the initial setting values for the inner, right outer, and left outer cameras.

**Table 6-21. Specifying the White Balance Standard Region**

| Parameter | Setting | Setting Value Range | IN1 | OUT1 | OUT2 |
|---|---|---|---|---|---|
| *startX* | Starting coordinate (horizontal) | 0 to 600 (in 40-pixel increments) | 0 | 0 | 0 |
| *startY* | Starting coordinate (vertical) | 0 to 450 (in 30-pixel increments) | 0 | 0 | 0 |
| *width* | Width | 40 to 640 (in 40-pixel increments) 640 or less when summed with *startX* | 640 | 640 | 640 |
| *height* | Height | 30 to 480 (in 30-pixel increments) 480 or less when summed with *startY* | 480 | 480 | 480 |

**Note:** The automatic adjustment of the white balance may be degraded if there is little change in the input images, or the input image has little contrast, as with a plain wall.

## Photo Mode

Call the nn::camera::SetPhotoMode() function to set the camera's photo mode to match the subject of the photo. Choose from the following values.

**Table 6-22. Photo Mode**

| Setting Value | Photo Mode | Description |
|---|---|---|
| PHOTO_MODE_NORMAL | No correction | No compensation is made to the camera settings. |
| PHOTO_MODE_PORTRAIT | Portrait mode | Settings are configured for portrait photography. |
| PHOTO_MODE_LANDSCAPE | Landscape mode | Settings are configured for landscape photography. |
| PHOTO_MODE_NIGHTVIEW | Night view mode | Settings are configured for photography in limited light conditions. |
| PHOTO_MODE_LETTER | Text mode | Settings configured for photographing a QR code pattern or other written characters. |

Changing the photo mode overwrites the contrast, gain, sharpness, exposure, and white balance settings as follows, and changes the standard regions for exposure for the outer and inner cameras. When set to **All**, the start coordinate is (0, 0) with a height of 480 and a width of 640. When set to **Center**, the start coordinate is (80, 60) with a height of 360 and a width of 480.

**Table 6-23. Photography Environment Settings Changed by Changing the Photo Mode**

| Photo Mode | Contrast | Gain | Sharpness | Exposure | White Balance | Outer Cameras | Inner Camera |
|---|---|---|---|---|---|---|---|
| No correction | NORMAL | Normal | 0 | 0 | NORMAL | All | Center |
| Portrait mode | LOW | Normal | -2 | 0 | NORMAL | Center | Center |
| Landscape mode | NORMAL | Normal | +1 | 0 | DAYLIGHT | All | Center |
| Night view mode | NORMAL | Maximum | –1 | +2 | NORMAL | All | Center |
| Text mode | HIGH | Normal | +2 | +2 | NORMAL | All | Center |

### Flipping

Call the `nn::camera::FlipImage()` function to specify how to flip the camera's output images. Choose from the following values.

**Table 6-24. Flipping**

| Setting Value | Flipping to Apply |
|---|---|
| FLIP_NONE | No flipping |
| FLIP_HORIZONTAL | Horizontal flipping |
| FLIP_VERTICAL | Vertical flipping |
| FLIP_REVERSE | Rotate image 180° |

### Effects

Call the `nn::camera::SetEffect()` function to specify any special effects to apply to the camera's output images. Choose from the following values.

**Table 6-25. Effects**

| Setting Value | Effect |
|---|---|

| | |
|---|---|
| EFFECT_NONE | No special effects |
| EFFECT_MONO | Monochrome |
| EFFECT_SEPIA | Sepia tone (ocher) |
| EFFECT_NEGATIVE | Negative |
| EFFECT_NEGAFILM | Film-tone negative Same as EFFECT_NEGATIVE, but with the U and V values swapped |
| EFFECT_SEPIA01 | Sepia tone (red ocher) |

Applying an effect and then changing other settings may change the effect. To give an image a softer feel, apply a sepia effect and then reduce the image's sharpness; to give an image a warmer, redder look, apply the film-tone negative effect and then raise the image's color temperature.

### Contrast

Call the `nn::camera::SetContrast()` function to set the camera's contrast (gamma curve). Choose from the following values.

**Table 6-26. Contrast**

| Setting Value | Contrast |
|---|---|
| CONTRAST_PATTERN_*n* (*n* is 01 to 11) | Contrast pattern number *n*. |
| CONTRAST_HIGH | Sets the contrast ratio to higher than the default value (pattern number 7). |
| CONTRAST_NORMAL | Default setting (pattern number 6). |
| CONTRAST_LOW | Sets the contrast ratio to lower than the default value (pattern number 5). |

### Lens Correction

Lens correction is a means of adjusting differences in brightness that may occur between the center and edges of an image by raising the brightness of the edges to more closely match the center. Call the `nn::camera::SetLensCorrection()` function to set the camera's lens correction. Choose from the following values.

**Table 6-27. Lens Correction**

| Setting Value | Lens Correction |
|---|---|
| LENS_CORRECTION_OFF | Disables lens correction. |
| LENS_CORRECTION_ON_70 | Enables lens correction at a value of 70. |
| LENS_CORRECTION_ON_90 | Enables lens correction at a value of 90. |
| LENS_CORRECTION_DARK | Sets edges to be darker than the default setting (LENS_CORRECTION_OFF). |
| LENS_CORRECTION_NORMAL | Default setting (LENS_CORRECTION_ON_70). |
| LENS_CORRECTION_BRIGHT | Sets edges to be brighter than the default setting (LENS_CORRECTION_ON_90). |

### Context

Use contexts to switch multiple photography environment settings at the same time. Use the `nn::camera::SwitchContext()` function to switch contexts. Each camera has two contexts, A and B, for a total of six sets of settings. These contexts can be switched independently, so the outer camera can be in context A, while the inner camera is in context B.

Each context can specify three settings: resolution, flipping, and special effects.

#### Noise Filter

When the screen changes from bright to dark or from dark to bright, the camera module applies a noise filter to remove noise from the automatically captured image. The images captured from one of the outer cameras may appear fuzzy when in stereo mode with this noise filter feature enabled. To prevent this, call the `nn::camera::SetNoiseFilter()` function and pass `false` for the *on* parameter to disable the noise filter. Pass `true` for the *on* parameter to enable the noise filter. All cameras have the noise filter enabled by default.

#### Configuring Photography Environment Settings in Batch

The API provides a feature to simultaneously configure all of the photography environment settings.

Call the `nn::camera::SetPackageParameterWithoutContext()` function to configure all photography environment settings with no context value specified (other than resolution, flipping, and effects).

Call the `nn::camera::SetPackageParameterWithContext()` function to configure all photography environment settings with context values specified (resolution, flipping, and effects).

## 6.4.3. Capture Settings

The images captured by the camera are written line by line in FIFO order, with the YUV-format data sent to a buffer prepared by the application. The application then sends the data to the YUVtoRGB circuit and the resulting RGB-format data to a buffer. The following diagram shows this set of operations.

**Figure 6-13. Captured Image Data Flow**



The following sections describe the settings required to take YUV-format image data through to the application buffer.

#### About Ports

As with the stereo mode, it is possible to use two cameras at the same time, making it necessary to specify which camera to use for acquiring captured images. Of the three cameras, the inner camera and the right outer camera are connected to one port, and the left outer camera is connected to its own port.

Most functions that configure image capture require that a port be specified, using the following enumerators.

**Table 6-28. Port Specifications**

| Enumerator | Description |
|---|---|
| PORT_NONE | No port is specified. |
| PORT_CAM1 | Specifies the port to which the inner and right outer cameras are connected. |
| PORT_CAM2 | Specifies the port to which the left outer camera is connected. |
| PORT_BOTH | Specifies both ports. |

### Trimming

Trimming crops a captured image to the size required by the application. Trim an image if the camera's resolution and the required size of the captured image differ.

Call the `nn::camera::SetTrimming()` function to enable or disable trimming. Call the `nn::camera::SetTrimmingParams` or `nn::camera::SetTrimmingParamsCenter()` functions to specify the range to trim. Call the `nn::camera::IsTrimming()` function to check whether trimming is currently enabled, and call the `nn::camera::GetTrimmingParams()` functions to get the trimming range.

> **Warning:** Set trimming prior to capturing images.

When setting the trimming position and range using the `nn::camera::SetTrimmingParams()` function, specify the position to start trimming as (x1, y1) and the position to stop trimming as (x2, y2). The (x1, y1) coordinate is included in the trimming operation, but the (x2, y2) coordinate is excluded. The following limitations apply to these settings.

- The x1 and y1 coordinate values for the trimming start position must be even values.
- x1 must be less than x2, and y1 must be less than y2.
- The post-trimming width (x2 – x1) and height (y2 – y1) must also be even values.
- The post-trimming image width times the height must be a multiple of 128.

When setting the trimming position and range using the `nn::camera::SetTrimmingParamsCenter()` function, specify the trimming size width and height (`trimWidth` and `trimHeight`) and the camera resolution width and height (`camWidth` and `camHeight`) for trimming based on the center of the captured image. The following code provides an example of how the (x1, y1) and (x2, y2) coordinates are calculated. Make sure that the calculated positions meet the restrictions described for the `SetTrimmingParams()` function.

**Code 6-16. Calculating Positions for Centered Trimming**

```
x1 = (camWidth - trimWidth) / 2;
y1 = (camHeight - trimHeight) / 2;
x2 = x1 + trimWidth;
```

```
y2 = y1 + trimHeight;
```

The captured image size sent to the buffer is the trimmed size. The cost of a transfer is still the same for different camera resolutions, provided you trim the images to the same size. However, trimming high-resolution images may cause the image to appear compressed due to the narrower field of view.

#### Number of Transfer Bytes (Lines)

The images captured by the camera are stored by the hardware FIFO line by line, and then written with multiple transfers to the buffer prepared by the application. The FIFO capacity is fixed at 10 KB. The number of bytes sent in a single transfer must meet the following conditions.

- The number of bytes sent in a single transfer must be a multiple of 256.
- The number of bytes sent in a single transfer must be no more than 10 KB (10,240 bytes).
- The total number of transfer bytes must be a multiple of the number of bytes sent in a single DMA transfer.

The total number of transfer bytes is the width times the height of the trimmed image, multiplied by 2, which is the number of bytes per pixel.

Call the `nn::camera::SetTransferLines()` function to set the number of bytes in a single transfer in terms of the number of lines. You can simply set the single-transfer line count to the value returned by the `nn::camera::GetMaxLines()` function, but note that execution halts in this function if the number of lines does not match the conditions described above. If you cannot set the number of lines, such as when setting an optional resolution using the `nn::camera::SetDetailSize()` function, instead call the `nn::camera::SetTransferBytes()` function to set the number of bytes in a single transfer. You can simply set the single-transfer byte count to the value returned by the `nn::camera::GetMaxBytes()` function, but note that execution halts in this function if the number of bytes (width × height × 2) is not a multiple of 256.

> **Warning:** Set the number of transfer bytes (lines) prior to capturing images. To prevent buffer errors, the `GetMaxLines()` and `GetMaxBytes()` functions calculate FIFO capacity as 5 KB.

Call the `nn::camera::GetTransferBytes()` function to get the current balance setting.

#### Receive Buffer

Call the nn::camera::GetFrameBytes function to get the size of the buffer required for receiving one frame of captured video. The starting address of the buffer is 4-byte aligned. An alignment below 64 bytes may reduce the transfer rate. **Only a buffer allocated in device memory may be specified.**

## 6.4.4. Starting Capture

Before transferring or capturing an image, call the `nn::camera::Activate()` function to activate

the camera to use for capture. You cannot activate the inner camera and the outer camera (R) at the same time, because the two cameras are connected to the same port. The outer camera (L) is connected to a different port. You can activate it at the same time as either the inner camera or the outer camera (R).

After making sure that the camera is active, call `nn::camera::SetReceiving` to begin transfer, and then call `nn::camera::StartCapture` to begin image capture.

**Code 6-17. nn::camera::SetReceiving Function**

```
void nn::camera::SetReceiving(nn::os::Event* pEvent, void* pDst,
        nn::camera::Port port, size_t imageSize, s16 transferUnit);
```

Specify the event for receiving notification that transfer has completed in the *`pEvent`* parameter. Specify the starting address of the receiving buffer to *`pDst`* aligned to `nn::camera::BUFFER_ALIGNMENT` (4 bytes). Specify the port in *`port`*. Specify the byte size of a single frame of captured video (the receive buffer size) in *`imageSize`*. In *`transferUnit`*, specify the byte size of the data in a single transfer as returned by `nn::camera::GetTransferBytes`.

You can call `nn::camera::IsFinishedReceiving` to get whether the single frame of captured video has finished transferring.

To check whether an error has occurred during capture, such as during FIFO writing, check whether the error event returned by `nn::camera::GetBufferErrorInterruptEvent` is in the signaled state. If a buffer error has occurred, this error event is in the signaled state and is an automatically resetting event of the `nn::os::Event` class. This error event also changes to a signal state when a camera error causes the camera to restart. Recover from an error by restarting in the order of transfer, and then capture.

You can call `nn::camera::IsBusy` to find out whether the camera is currently capturing an image. You can also call `nn::camera::GetVsyncInterruptEvent` to get the event that enters the signaled state when the camera's VSYNC interrupt occurs. Use this function in a process that synchronizes with camera VSYNC, or in a process that changes the camera environment when a frame is not being transferred.

Call `nn::camera::SynchronizeVsyncTiming` to synchronize the camera VSYNC interrupt timing, such as when you are using the stereo camera to display in stereoscopic view. This function attempts to synchronize the VSYNC interrupt timing of the two cameras. This process does not maintain the synchronization. The function must be called again when synchronization is lost. The four frames after calling this function may be very dark, because it places the camera into standby before restarting it. The two cameras have a small gap in the timing of the image capture being sent, even if the cameras have the same configuration. VSYNC interrupt timing has a large gap after calling `nn::camera::Activate` to start the camera, or when changing the camera configuration even when capturing video with the same configuration and fixed frame rate.

The capture image (YUV4:2:2 format) is output in the following alignment.

| + 0 Byte | + 1 Byte | + 2 Byte | + 3 Byte |
|---|---|---|---|
| Y (n) | U (n) | Y (n + 1) | V (n) |

## 6.4.5. YUVtoRGB Circuit Settings

Use the YUVtoRGB circuit to convert YUV-format data to RGB format at the hardware level. This circuit also supports output to the native GPU block format. However, the CTR system is only equipped with one YUVtoRGB circuit. When converting captured images from multiple cameras,

such as when in stereo mode, use a mutex or other mechanism to make sure that multiple conversion requests are not issued simultaneously.

This section describes settings related to data conversion using the YUVtoRGB circuit.

### Input Format

Call the `nn::y2r::SetInputFormat()` function to set the input YUV data format. Call the `nn::camera::GetInputFormat()` function to get the current setting. Choose from the following values.

**Table 6-29. Input Format**

| Setting Value | Format |
|---|---|
| `INPUT_YUV422_INDIV_8` | Input the individual Y, U, and V values for YUV4:2:2 as 8 bits each. |
| `INPUT_YUV420_INDIV_8` | Input the individual Y and U values for YUV4:2:0 as 8 bits each. |
| `INPUT_YUV422_INDIV_16` | Input the individual Y, U, and V values for YUV4:2:2 as 16 bits each (padding required). |
| `INPUT_YUV420_INDIV_16` | Input the individual Y and U values for YUV4:2:0 as 16 bits each (padding required). |
| `INPUT_YUV422_BATCH` | Input the Y, U, and V values for YUV4:2:2 all together in 32 bits. |

The data formats for each value are as follows.

YUV4:2:2/YUV4:2:0 individual input (8 bit)

| Component | + 0 Byte | + 1 Byte | + 2 Byte | + 3 Byte |
|---|---|---|---|---|
| Y | Y (n) | Y (n + 1) | Y (n + 2) | Y (n + 3) |
| U | U (n) | U (n + 1) | U (n + 2) | U (n + 3) |
| V | V (n) | V (n + 1) | V (n + 2) | V (n + 3) |

YUV4:2:2/YUV4:2:0 individual input (16 bit padding)

| Component | + 0 Byte | + 1 Byte | + 2 Byte | + 3 Byte |
|---|---|---|---|---|
| Y | Y (n) | padding | Y (n + 1) | padding |
| U | U (n) | padding | U (n + 1) | padding |
| V | V (n) | padding | V (n + 1) | padding |

YUV4:2:2 batch input

| Component | + 0 Byte | + 1 Byte | + 2 Byte | + 3 Byte |
|---|---|---|---|---|
| YUV | Y (n) | U (n) | Y (n + 1) | V (n) |

You can only get captured images from the camera in YUV batch format. In most cases, specify `INPUT_YUV422_BATCH` when converting captured images.

### Line Width and Line Count

Call the `nn::y2r::SetInputLineWidth()` function to specify the line width of the data to

convert (the input data). Call the `nn::y2r::GetInputLineWidth()` function to get the current setting. The line width must be set to a multiple of 8, up to a maximum value of 1024.

Call the `nn::y2r::SetInputLines()` function to set the number of input lines. Call the `nn::y2r::GetInputLines()` function to get the current setting.

### Output Format

The data converted by the YUVtoRGB circuit is stored in the output buffer. Call the `nn::y2r::SetOutputFormat()` function to set the output data format. Call the `nn::camera::GetOutputFormat` function to get the current setting. Choose from the following output formats.

**Table 6-30. Output Format**

| Setting Value | Format |
|---|---|
| `OUTPUT_RGB_32` | 32-bit RGB (RGBA8888) |
| `OUTPUT_RGB_24` | 24-bit RGB (RGB888) |
| `OUTPUT_RGB_16_555` | 16-bit RGB (RGBA8888) |
| `OUTPUT_RGB_16_565` | 16-bit RGB (RGB565) |

The data formats for each value are as follows.

32-bit RGB (`OUTPUT_RGB_32`)

24-bit RGB (`OUTPUT_RGB_24`)

16-bit RGB (`OUTPUT_RGB_16_555`)

16-bit RGB (`OUTPUT_RGB_16_565`)

Formats that output the alpha component use the alpha value that was set by the `nn::y2r::SetAlpha()` function. The `OUTPUT_RGB_32` format uses bits 0 to 7 of the set alpha value, while the `OUTPUT_RGB_16_555` format only uses the 7th bit. Call the `nn::camera::GetAlpha()` function to get the current alpha value.

### Block Alignment

Call the `nn::y2r::SetBlockAlignment()` function to set the block alignment for the data stored in the output buffer. Call the `nn::camera::GetBlockAlignment()` function to get the current setting. Choose from the following block alignments.

**Table 6-31. Block Alignment**

| Setting Value | Block Alignment |
|---|---|
| `BLOCK_LINE` | Horizontal line format. The usual linear format. The 24-bit and 32-bit RGB output formats cannot be used directly as OpenGL standard-format textures due to byte-order issues. |
| `BLOCK_8_BY_8` | 8x8 block format. This is the native GPU block format. Data in this format can be used as native-format textures. |

**Warning:** When using `BLOCK_8_BY_8` block alignment, the input image height (vertical line count) must be a multiple of 8.

### Output Buffer

You can call the `nn::y2r::GetOutputImageSize()` function to get the size of the buffer required for receiving one frame of output data. The starting address of the buffer is 4-byte aligned. An alignment below 64 bytes may reduce the transfer rate. **Only a buffer allocated in device memory may be specified.**

### Conversion Coefficient

Choose the coefficient for YUV to RGB conversion from the following standard conversion coefficients. Call the `nn::y2r::SetStandardCoefficient()` function to set the conversion coefficient.

Select the type of conversion coefficient for converting images output by the cameras from those returned by the `nn::camera::GetSuitableY2rStandardCoefficient()` function. When making a selection, consider the possibility of future changes in the camera module.

You can choose from the following four types of standard conversion coefficients.

**Table 6-32. Types of Standard Conversion Coefficient**

| Setting Value | Conversion Coefficient Type (Value Range) |
|---|---|
| `COEFFICIENT_ITU_R_BT_601` | ITU-R BT.601 (0 ≤ Y, U, V ≤ 255) |
| `COEFFICIENT_ITU_R_BT_709` | ITU-R BT.709 (0 ≤ Y, U, V ≤ 255) |
| `COEFFICIENT_ITU_R_BT_601_SCALING` | ITU-R BT.601 (16 ≤ Y ≤ 235, 16 ≤ U, V ≤ 240) |
| `COEFFICIENT_ITU_R_BT_709_SCALING` | ITU-R BT.709 (16 ≤ Y ≤ 235, 16 ≤ U, V ≤ 240) |

### Rotation

You can rotate an image when converting the format. To specify how many degrees to rotate an image, call the `nn::y2r::SetRotation()` function; to get the current rotation degree setting, call the `nn::y2r::GetRotation()` function. You can choose from the following four rotation settings, including no rotation.

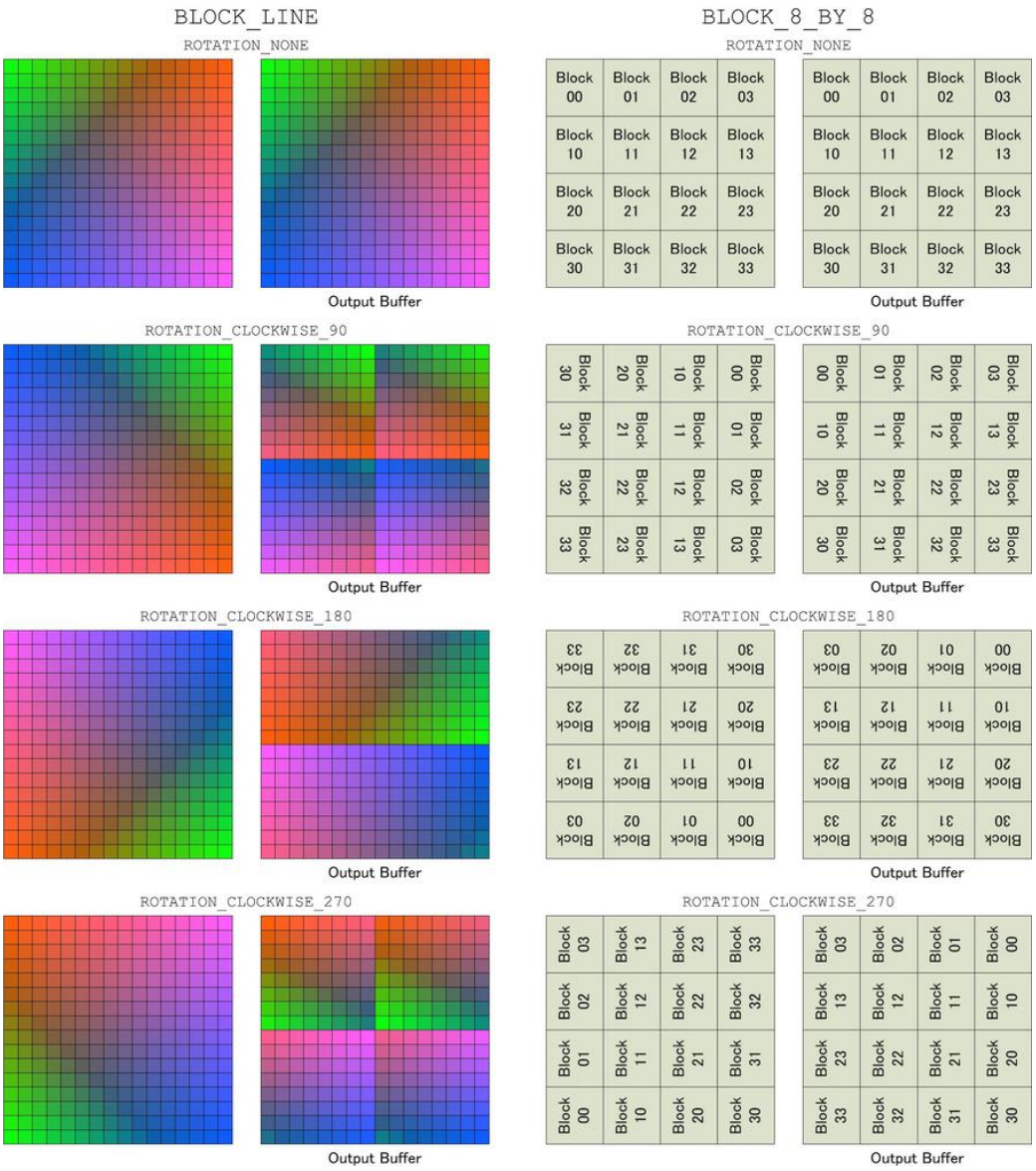
**Table 6-33. Rotation Angle**

| Setting Value | Rotation Angle |
|---|---|
| ROTATION_NONE | No rotation. |
| ROTATION_CLOCKWISE_90 | Clockwise 90° |
| ROTATION_CLOCKWISE_180 | 180° |
| ROTATION_CLOCKWISE_270 | Clockwise 270° (counterclockwise 90°) |

When you rotate an image, the post-conversion data no longer has valid image data ordering. Consequently, your application must correct the data ordering after receiving each frame.

**Note:** This may be changed in the future so that valid data ordering is output by transfer.

**Figure 6-14. Output Data Ordering Differences Due to Block Alignment and Rotation**




## Batch Configuring

Call the `nn::y2r::SetPackageParameter()` function to configure all of the YUVtoRGB circuit settings at the same time. You can get all the settings at the same time using the `nn::y2r::GetPackageParameter()` function.

## 6.4.6. Starting Format Conversion

Data conversion is carried out in parallel with sending and receiving, so preparations for sending and receiving data must be completed before conversion is begun.

Use the following functions to prepare to send input data. `nn::y2r::SetSendingYuv` for batched YUV data. `nn::y2r::SetSendingY` for just the Y data. `nn::y2r::SetSendingU` for just the U data. `nn::y2r::SetSendingV` for just the V data. These functions take the input data buffer's size, the total transfer data size, and the size of one line of input data, specified in bytes, as arguments. **Only a buffer allocated in device memory may be specified.** Note that the total transfer data size must be a multiple of the size of one line. The offset value added when one line of input data is transferred (*`transferStride`*) may be specified for any of these functions.

Call `nn::y2r::SetReceiving` to prepare to receive output data. Specify the buffer for storing output data. (**VRAM buffer cannot be specified. Only a buffer allocated in device memory may be specified. Again, you must align it to `nn::y2r::BUFFER_ALIGNMENT` (4 bytes).**) Specify the total size of received data and the received data size per transfer in byte units. To improve performance, we recommend specifying the transfer size as the size of eight lines of output data. Call the `nn::camera::GetOutputImageSize()` function to get the current setting. The size value specified for eight lines must be the size of a single pixel in bytes returned by the `nn::y2r::GetOutputFormatBytes()` function multiplied by the width of a single line multiplied by 8. You can also specify an offset value added to each transfer (*`transferStride`*). To add an offset to each line, specify the size of one line of output data for the size of a single transfer.

A good time to prepare to send and receive data is immediately after the event specified by the `nn::camera::IsFinishedReceiving` or `nn::camera::a SetReceiving function` is in the signaled state, which indicates that the captured image has been fully received and you have confirmed the input data. Before preparing to send or receive data, call the `nn::y2r::IsBusyConversion()` function to check whether the system is busy converting formats.

Call the `nn::y2r::StartConversion()` function to begin format conversion after data send/receive preparations are complete. Data is sent and received at the same time as format conversion begins. Call the `nn::y2r::StopConversion()` function if you must forcibly halt format conversion, such as when an error occurs during capture operations.

Call the `nn::y2r::IsBusyConversion()` function to check whether the system is busy converting formats. To check whether the system has completed sending input data, call the `nn::y2r::IsFinishedSendingYuv()` function for batched YUV data, the `nn::y2r::IsFinishedSendingY()` function for just the Y data, the `nn::y2r::IsFinishedSendingU()` function for just the U data, and the `nn::y2r::IsFinishedSendingV()` function for just the V data. To check whether the system has finished receiving output data, call the `nn::y2r::IsFinishedReceiving()` function.

You can use a combination of these functions to check whether format conversion and data transmission has completed for a frame. You can also check whether data transfer is complete by using the event class (automatically resetting event) returned by the `nn::y2r::GetTransferEndEvent()` function. This event class receives an interrupt notification when transfer has completed. The `nn::y2r::SetTransferEndInterrupt()` function must be called to enable interrupt notifications before getting the event class. (The default is to disable them.) Call the `nn::y2r::GetTransferEndInterrupt()` function to check whether notifications

are currently enabled.

A hardware bug prevents the transfer completion event obtained by the `nn::y2r::GetTransferEndEvent()` function from being signaled, and prevents the `nn::y2r::IsBusyConversion()` function from returning `true`, if the following (rare) sequence occurs: (1) while using the cameras and YUVtoRGB circuit simultaneously, (2) a buffer error occurs during camera data transfer, and (3) during recovery from that error the processing of the `nn::camera::SetReceiving()` function overlaps the YUVtoRGB circuit data transfer processing at a particular time. For this reason, always make sure to specify a timeout when waiting for the signal status of a transfer completion event.

Specify a timeout longer than the time required for conversion. The time required for conversion depends on the size of the input image and the output format. Output of a VGA image in 16-bit RGB format takes about 13 milliseconds. Output of a VGA image in 24-bit RGB format takes about 16 milliseconds.

If this bug occurs, forcibly terminate conversion by calling the `nn::y2r::StopConversion()` function. This makes the YUVtoRGB circuit usable again.

The probability of occurrence of this bug is proportional to the frequency of camera buffer errors, so use the camera under conditions where buffer errors are unlikely to arise. When buffer errors occur frequently, respond by lowering the camera frame rate or with another method. Also, set a higher priority for the thread that calls the `nn::camera::SetReceiving()` function.

## 6.4.7. Playing the Shutter Sound

The shutter sound is forcibly played even when the speaker volume is set to 0 and the camera LED is temporarily turned off.

> **Note:** SNAKE and FTR do not have a camera indicator LED.

**Code 6-18. Playing the Shutter Sound**

```
nn::Result nn::camera::PlayShutterSound(nn::camera::ShutterSoundType type);
```

Specify one of the following shutter sound types for the *type* parameter.

**Table 6-34. Shutter Sound Types**

| Setting Value | Shutter Sound Type |
|---|---|
| SHUTTER_SOUND_TYPE_NORMAL | Normal shutter sound. |
| SHUTTER_SOUND_TYPE_MOVIE | Sound played when starting video capture. |
| SHUTTER_SOUND_TYPE_MOVIE_END | Sound played when ending video capture. |

## 6.4.8. Ending Capture

To stop image capture, do the following. Stopping image capture without completing the following steps can lead to audio noise when displaying the HOME Menu.

1. Call the `nn::y2r::StopConversion()` function to stop format conversion.
2. Call the `nn::camera::StopCapture()` function to stop capture.
3. Call `nn::camera::Activate(SELECT_NONE)` to put all cameras into standby mode.
4. Call the `nn::camera::Finalize()` and `nn::y2r::Finalize()` functions to close the camera and Y2R libraries.

## 6.4.9. Support for Sleep Mode

`nn::camera::ResultIsSleeping` (or `nn::y2r::ResultIsSleeping`) is returned to indicate the system is closed when a `Camera` (or Y2R) library function is called while the system is closed. This includes during initialization. Note that this return value is returned as long as the system is closed, even if it is not in Sleep Mode.

The camera is designed so that operations stop if the system is closed, regardless of Sleep Mode. If the system is closed while capturing an image, capture resumes when the system is opened. Note, however, that when capture resumes there is a period of time immediately after the camera is activated by the `Activate()` function where the image is unstable. This is due to processing equivalent to the `nn::camera::Activate(SELECT_NONE)()` function being performed inside the library when the system is closed. There is also a possibility that the system will enter a state where the `IsBusy()` function always returns `true` depending on the exact timing the system was closed. This status will be canceled when the system is opened, but do not perform polling using the `IsBusy()` function if a process for entering Sleep Mode when the system is closed has been implemented.

If the system enters sleep in the midst of RGB conversion by the Y2R library, conversion is forcibly terminated. Conversion does not resume after recovering from sleep. If supporting sleep, do not enter sleep during conversion (when the `IsBusyConversion()` function returns `true` or while waiting for an event obtained by the `GetTransferEndEvent()` function). Implement code so that the system only enters sleep after checking that conversion is complete. Particularly note that events will not enter signal status after recovering from sleep if sleep is entered while waiting for an event obtained using the `GetTransferEndEvent()` function.

## 6.4.10. Conflicts With Camera Mode

You cannot press the L Button and the R Button simultaneously to enter camera mode from the HOME Menu while an application is using the cameras. If you attempt to do so, a dialog box is displayed with a message stating that the cameras are in use.

The cameras are considered to be in use if either the CAMERA or the Y2R library has been initialized. If an application that uses the cameras initializes either of these libraries when it is started, the cameras are considered to be in use when the HOME Menu is displayed even if the application is not actually using them. Even if the Y2R library alone is used to convert YUV images into RGB images—during movie playback, for example—the cameras are considered to be in use and the system does not enter camera mode.

Initialize the CAMERA and Y2R libraries just before you use them and shut them down afterwards whenever you can. Do not leave the CAMERA and Y2R libraries initialized while they are not in use.

# 7. File System

This chapter describes the FS library used to access media and cautions for each media type.

## 7.1. FS Library

You must use the FS library to access 3DS system memory devices (Nintendo 3DS Cards and SD cards).

### 7.1.1. Initializing

Call the `nn::fs::Initialize()` function to initialize the FS library. It is not a problem to call this function again even after the library is already initialized.

Function calls against files or directories cause an error unless they are made after the FS library is initialized.

### 7.1.2. Finalizing

To finish using the FS library, close all open files and directories, and unmount all archives.

You must also either finalize or cancel use of the FS library when shutting down the application and when transitioning to Sleep Mode. For more information, see 5.3.1.1. Cautions When Shutting Down and 5.3.3.4. Prohibited Processes While in Sleep Mode.

### 7.1.3. Specifying Paths

All paths (to files or directories) must be specified as **absolute paths**. The "/" symbol (slash) is used as the path delimiter.

You can use wide or multiple-byte characters (ASCII characters only) in path specification strings used with FS library functions. However, keep track of the stack size when using multiple-byte strings because the library must convert multiple-byte strings to wide-character strings, which requires the allocation of a large buffer from the stack. Consequently, use wide-character strings unless you have a good reason not to.

### 7.1.4. Accessing Files

Choose the best class for your purposes from the following three types for accessing files on media.

**Table 7-1. Classes Used for File Access**

| Class | Description |
|---|---|
| `nn::fs::FileInputStream` | Opens a file for reading. |
| `nn::fs::FileOutputStream` | Opens a file for writing. If the specified file does not exist, one may be created depending on the settings specified in the class initialization. |
| `nn::fs::FileStream` | Opens a file for reading and writing, depending on the access mode specified. |

**Warning:** Much as with `Initialize()` and `TryInitialize()`, the functions used for file access have versions prepended with and without `Try`. Both versions operate the same way, but make sure to use the `Try` versions within your application.

FS library functions generally do not return until they complete their execution. The library also processes file-access requests in the order they arrive, so a file access by a high-priority thread will not be processed until the file accesses requested before it have completed.

## 7.1.4.1. nn::fs::FileInputStream Class

Use this class to open a file for reading. Specify the file for reading in the arguments to the constructor or to the `Initialize` or `TryInitialize()` functions. The application halts if the file specified in the arguments to the constructor or to the `Initialize()` function does not exist. Use the return value from the `TryInitialize()` function instead for error checking. The application halts if file access is attempted from an instance that did not successfully open the file or if the file is opened again from an instance that already has it open.

Use the `GetSize` or `TryGetSize()` functions to get the size of the file. This can be used to determine the size to use for the read buffer.

Use the `Read` or `TryRead()` functions to read a file. The arguments specify the buffer into which to copy file contents, and the size of the buffer. The return value is the number of bytes actually copied to the buffer, or `0` if the end of the file has been reached. If at all possible, allocate the buffer with a 4-byte alignment. Though dependent on device and memory region, read speeds are substantially slower for buffers that are not 4-byte aligned and whose current position does not change in 4-byte units.

Use the `GetPosition` or `TryGetPosition()` functions to get, or the `SetPosition` or `TrySetPosition()` functions to set, the file read start position (current position). The current position is the number of bytes from the start of the file. You can also use the `Seek` or `TrySeek()` functions to set the base position and the offset from there. Note that although read speeds are dependent on device and memory region, read speeds are substantially slower if the current position is not set to a multiple of 4 bytes from the start of the file. If the current position is set before the start of the file or after the end of the file, the application halts on the next attempt at accessing the file.

**Table 7-2. Specifying the Base Position**

| Setting Value | Description |
|---|---|
| `POSITION_BASE_BEGIN` | Sets the current position based on the start of the file. |
| `POSITION_BASE_CURRENT` | Sets the current position based on the current position in the file. |

| | |
|---|---|
| POSITION_BASE_END | Sets the current position based on the end of the file. |

Call the Finalize() function to close the file after you have finished using it.

The following code sample shows opening a file using TryInitialize, checking whether it opened properly, and then reading from it.

**Code 7-1. File Reading**

```
nn::fs::FileInputStream fis;
nn::Result result = fis.TryInitialize(L"rom:/test.txt");
if (result.IsSuccess())
{
    s64 fileSize;
    result = fis.TryGetSize(&fileSize);
    NN_LOG("FileSize=%lld\n", fileSize);
    buf = heap.Allocate(fileSize);
    s32 ret;
    result = fis.TryRead(&ret, buf, fileSize);
    ...
    heap.Free(buf);
}
fis.Finalize();
```

## 7.1.4.2. nn::fs::FileOutputStream Class

Use this class to open a file for writing. Specify the file for writing in the arguments to the constructor or to the Initialize or TryInitialize() functions. Use the return value from the TryInitialize() function instead for error checking. If the specified file does not exist and the *createIfNotExist* parameter is set to true, a new file of size zero is created. The application halts if the file specified in the arguments to the constructor or to the Initialize() function does not exist and the *createIfNotExist* parameter is set to false. The application halts if file access is attempted from an instance that did not successfully open the file or if the file is opened again from an instance that already has it open.

Use the SetSize() or TrySetSize() functions before writing to set the size of the file. The file writing position will be adjusted if the file size is reset smaller in the middle of a write operation. The GetSize or TryGetSize() functions return the current file size.

**Note:** The TryRead() function successfully reads in the file of size zero created by the TryInitialize() function. You must be careful if the TryInitialize() function is called with true passed to the *createIfNotExist* parameter, and then later the file size is set using the TrySetSize() function.

When the Game Card is removed or the process is otherwise interrupted between calls to TryInitialize() and TrySetSize(), the next time TryRead() is called, the result is that a file of size zero is read.

For handling files of fixed size, if the application is not checking the size using the TryGetSize() function before calling TryRead, we recommend creating the file using the TryCreateFile() function, which can set the file size, rather than using the *createIfNotExist* parameter to create the file.

Use the Write or TryWrite() functions to write data to the file. The arguments specify the starting address of a buffer that contains data to write to the file and the number of bytes of data

to write. The return value is the number of bytes actually written to the file. When writing past the end of the file, the file is expanded if possible. If at all possible, allocate the buffer with a 4-byte alignment. Though dependent on device and memory region, write speeds are substantially slower for buffers that are not 4-byte aligned. You can specify whether to flush the file cache (write the cache contents to media) while the file is written to by passing the appropriate value in the *flush* parameter. We recommend specifying `false` for this value to avoid wearing out the memory media and to prevent data corruption from memory cards possibly being removed during write operations. Instead, call the `Flush` or `TryFlush()` functions just before closing the file. However, if you chose not to write and flush concurrently, make sure to flush the cache before you close the file.

Use the `GetPosition` or `TryGetPosition()` functions to get, or the `SetPosition` or `TrySetPosition()` functions to set, the file write start position (current position). The current position is the number of bytes from the start of the file. You can also use the `Seek` or `TrySeek()` functions to set the base position and the offset from there. Note that although write speeds are dependent on device and memory region, write speeds are substantially slower if the current position is not set to a multiple of 4 bytes from the start of the file. If the current position is set before the start of the file or after the end of the file, the application halts on the next attempt at accessing the file.

Call the `Finalize()` function to close the file after you have finished using it.

The following code sample shows opening a file using `TryInitialize`, checking if it opened properly, and then writing to it.

**Code 7-2. File Writing**

```
nn::fs::FileOutputStream fos;
nn::Result result = fos.TryInitialize(L"sdmc:/test.txt", true);
if (result.IsSuccess())
{
    s32 ret;
    result = fos.TryWrite(&ret, buf, sizeof(buf));
}
fos.Finalize();
```

## 7.1.4.3. nn::fs::FileStream Class

Use this class to open a file for both reading and writing. The initialization arguments are different, but the member functions operate the same way as for the previous two classes.

Specify the file to access and the access mode in the arguments to the constructor or to the `Initialize` or `TryInitialize()` functions. Specify the access mode as a combination of the following flags.

**Table 7-3. Access Mode Flags**

| Flag | Description |
|---|---|
| `OPEN_MODE_READ` | Opens a file for reading. |
| `OPEN_MODE_WRITE` | Opens a file for writing. Reading is also possible. |
| `OPEN_MODE_CREATE` | When combined with `OPEN_MODE_WRITE`, if the file specified at initialization does not already exist, it is created. |

## 7.1.5. Accessing Directories

Use the `nn::fs::Directory` class to get information about directories and directory contents (subdirectories and files).

### 7.1.5.1. nn::fs::Directory Class

Use this class to open directories and list their contents. Specify the directory in the arguments to the constructor or to the `Initialize` or `TryInitialize()` functions. The application halts if the directory specified in the arguments to the constructor or to the `Initialize()` function does not exist. Use the return value of the `TryInitialize()` function instead for error checking. The application halts if directory access is attempted from an instance that did not successfully open the directory, or if the directory is opened again from an instance that already has it open.

A slash is needed at the end of the path when specifying the root directory for media, such as the "`/`" at the end of the "`sdmc:/`" path. The ending slash may also be added for non-root directories, but it is not needed.

Use the `Read()` function to get directory entries. The arguments specify an array for directory entry information (an `nn::fs::DirectoryEntry` structure) and the number of elements in the array. The return value is the number of entries actually stored in the array. After all entries have been obtained, calls to the `Read()` function return `0`.

Call the `Finalize()` function to close the directory after you have finished using it.

The following code sample shows opening a directory using `TryInitialize`, checking whether it opened properly, and then getting its entries.

**Code 7-3. Directory Access**

```
nn::fs::Directory dir;
nn::fs::DirectoryEntry entry[ENTRY_MAX];
nn::Result result = dir.TryInitialize(L"sdmc:/TestDirectory");
if (result.IsSuccess())
{
    s32 readCount;
    while (true)
    {
        result = dir.TryRead(&readCount, entry, ENTRY_MAX);
        if (readCount == 0) break;
        ...
    }
}
dir.Finalize();
```

## 7.1.6. File and Directory Operations

Use these functions to create, rename, or delete files and directories.

### 7.1.6.1. Creating Files

Use the `nn::fs::CreateFile` or `nn::fs::TryCreateFile()` functions to create a file of the

specified size. The application halts if an error occurs on a call to the `nn::fs::CreateFile()` function. Use the return value from the `nn::fs::TryCreateFile()` function instead for error checking.

An error occurs if you attempt to create a file of the same full name as an existing file. If an error occurs for some reason other than during an attempt to create a file with the same name as an existing file, it is possible that an unnecessary file has been created, so delete any such unnecessary files. If no unnecessary files have been created, the delete operation returns `nn::fs::ResultNotFound`.

### 7.1.6.2. Renaming Files

Use the `nn::fs::RenameFile` or `nn::fs::TryRenameFile()` functions to rename a file. The application halts if an error occurs on a call to the `nn::fs::RenameFile()` function. Use the return value from the `nn::fs::TryRenameFile()` function instead for error checking.

An error occurs if you call these functions on a file that is open or if you attempt to rename a file to a name already used by another file in the same directory (including attempting to rename a file and specifying the name it already has).

### 7.1.6.3. Deleting Files

Use the `nn::fs::DeleteFile` or `nn::fs::TryDeleteFile()` functions to delete a file. The application halts if an error occurs on a call to the `nn::fs::DeleteFile()` function. Use the return value from the `nn::fs::TryDeleteFile()` function instead for error checking.

An error occurs if you call these functions on a file that is open.

### 7.1.6.4. Creating Directories

Use the `nn::fs::CreateDirectory` or `nn::fs::TryCreateDirectory()` functions to create a directory. The application halts if an error occurs on a call to the `nn::fs::CreateDirectory()` function. Use the return value from the `nn::fs::TryCreateDirectory()` function instead for error checking.

An error occurs if you call these functions on a directory located in a nonexistent directory or if you attempt to create a directory with a name already used by another directory in the same parent directory.

### 7.1.6.5. Renaming Directories

Use the `nn::fs::RenameDirectory` or `nn::fs::TryRenameDirectory()` functions to rename a directory. The application halts if an error occurs on a call to the `nn::fs::RenameDirectory()` function. Use the return value from the `nn::fs::TryRenameDirectory()` function instead for error checking.

An error occurs if you call these functions on a directory that is open or if you attempt to rename

a directory to a name already used by another directory in the same parent directory (including attempting to rename a directory and specifying the name it already has).

#### 7.1.6.6. Deleting Directories

Use the `nn::fs::DeleteDirectory` or `nn::fs::TryDeleteDirectory()` functions to delete a directory. A directory cannot be deleted unless it is empty. The application halts if an error occurs on a call to the `nn::fs::DeleteDirectory()` function. Use the return value from the `nn::fs::TryDeleteDirectory()` function instead for error checking.

An error occurs if you call these functions on a directory that is open.

The `nn::fs::TryDeleteDirectoryRecursively()` function attempts to completely delete the specified directory by recursively deleting all entries. If an error occurs during this process, the function returns that error.

### 7.1.7. Checking the SD Card State

The SDK provides functions to check whether an SD card is inserted, to send notification when an SD card is inserted or removed, and to check whether an SD card can be written to.

**Code 7-4. Checking the SD Card Insertion State, Notifying of Insertion/Removal, and Checking for Writability**

```
bool nn::fs::IsSdmcInserted();
void RegisterSdmcInsertedEvent(nn::os::LightEvent* p);
void UnregisterSdmcInsertedEvent();
void RegisterSdmcEjectedEvent(nn::os::LightEvent* p);
void UnregisterSdmcEjectedEvent();
bool nn::fs::IsSdmcWritable();
nn::Result nn::fs::GetSdmcSize(s64* pTotal, s64* pFree);
```

Call the `nn::fs::IsSdmcInserted()` function to check whether an SD card is inserted. The function returns `true` if a device is currently inserted in the SD card slot, even if it is a broken SD card or something other than an SD card, such as an empty SD card adapter. The processing to check whether an SD card is inserted entails a heavy processor load; we recommend using the following functions to register an event class that waits for notification when an SD card is inserted or removed.

Call the `nn::fs::RegisterSdmcInsertedEvent()` and `nn::fs::UnregisterSdmcInsertedEvent()` functions to register and unregister an instance of the `nn::os::LightEvent` class to receive notifications of SD card insertions.

Call `nn::fs::RegisterSdmcEjectedEvent` and `nn::fs::UnregisterSdmcEjectedEvent` to register and unregister an instance of the `nn::os::LightEvent` class to receive notifications of SD card ejections.

The `nn::fs::IsSdmcWritable()` function returns `true` if an SD card is inserted and can be written to.

The `nn::fs::GetSdmcSize()` function returns the total capacity of an SD card (*`pTotal`*) and the available capacity (*`pFree`*).

## 7.1.8. Latency Emulation

You can use the following function to debug changes in access speeds. It emulates file access latency in an application caused by conflicting file system access with SpotPass or another background process.

**Code 7-5. Initializing Latency Emulation**

```
void nn::fs::InitializeLatencyEmulation(void);
```

Use **Debug Mode** in the Config tool's **Debug Settings** to enable and disable latency emulation. While the application is running, file accesses are delayed by the number of milliseconds specified by **FS Latency Emulation** only when the corresponding item is set to **enable** in the Config tool and this function has enabled latency emulation.

## 7.1.9. Access Priority Settings

File system access priority is supported. This allows you to access multiple files from multiple threads, because the execution order is appropriately adjusted, and items with a higher priority setting are processed more quickly.

Access priority requires real-time capability in the same way as streaming (an established number of cycles is required for an established amount of processing). Appropriate settings are provided for file access. Using these settings keeps delay time resulting from file access to a minimum.

### 7.1.9.1. Types of Access Priority

The access priorities that can be set are shown in the following table, in priority order.

**Table 7-4. Access Priority Types**

| Type | Definition | Description |
|---|---|---|
| Real-time Priority | `PRIORITY_APP_REALTIME` | Special priority level used for accessing files which, if loading is delayed, could detract from the user's experience. This includes the loading of streaming data.<br>There are, however, some restrictions on its use. |
| Normal Priority | `PRIORITY_APP_NORMAL` | Priority level used for accessing general purpose files, such as model data, scene data, and all types of save data. |
| Low Priority | `PRIORITY_APP_LOW` | Priority level used for accessing files which can be executed anytime the system is available for processing, for which the priority is lower than normal. This includes autosave. |

The definition used to specify access priority level is defined in the `nn::fs::PriorityForApplication` enumerator.

> **Note:** For the restrictions on real-time priority see the *CTR-SDK API Reference*.

### 7.1.9.2. Access Priority Setting Targets

The following list shows a range of targets for access priority settings.

**Table 7-5. Access Priority Setting Targets and Functions Used in Settings**

| Setting Target | Function Used in Setting |
|---|---|
| Overall File System | `nn::fs::SetPriority` |
| Archive | `nn::fs::SetArchivePriority` |
| File Stream and Directory | `TrySetPriority` or `SetPriority` of all classes |

The *Overall File System* setting target refers to general access from the application and accessing of archives performed without specifying an archive name. This includes save data formats. If an access priority has not been set explicitly, the normal priority is applied.

The **Archive** setting target refers to archives whose name must be specified for an archive that is already mounted. Different access priority settings are possible for each archive. If the access setting is not set explicitly, the overall file system setting at the time the archive was mounted is applied. Even if the setting is changed for the overall system after the archive is mounted, the archive setting itself is not affected. Similarly, even if the archive setting is changed, the setting for the overall file system is not affected.

The *File Stream and Directory* setting is applied to objects accessed in the file stream (such as `nn::fs::FileStream`) and directory objects (`nn::fs::Directory`). Access priority settings can differ for each object even if it resides in the same file or directory. Unless the access priority setting is set explicitly, the archive setting at the point when the object was created is applied. Even if the archive setting is changed after creating an object, the various settings for each object is not affected. Similarly, even if individual settings of various objects are changed, the archive setting is not affected.

### 7.1.9.3. Cautions

Access priority changes the priorities within the file system. If the priority of the threads that call file access functions is not set at a high level, the file access process will not immediately start (even if a particular item has real-time priority). Set the priority of threads that call up file access functions at a high enough priority to meet the demands of their particular processing.

In addition, the order of completing access of desired files is not guaranteed. There is always a possibility that access of files with a lower priority level may be completed before files with a higher access priority, even though they were requested at a later time. Do not implement an application so that it is dependent on the order of file access completion, when multiple files are accessed in parallel.

Also, do not perform file access performance design based on measured performance values. For estimated access times, see *CTR-SDK API Reference*.

## 7.1.10. Restriction on the Number of Files and Directories Opened Simultaneously

There are limits to the number of files and directories that can be opened at the same time using the file system.

For safe operation, limit the number of files and directories that an application opens simultaneously to the following.

- Up to four files of the archive that directly access the SD card and the extended save data archive combined
- Up to 10 files for the save data archive (including the save data of other applications)
- Up to 10 total files

**Note:** A ROM archive is not subject to these limits. Its limits are in accordance with the parameters defined by calling the `MountRom()` function.

## 7.2. ROM Archives

ROM archives are read-only archives for accessing ROMFS that are created at build time. ROM archives must be mounted explicitly by applications. The mounting procedure and arguments are the same regardless of whether the application is card-based software or a downloadable application.

The use of ROM archives requires working memory allocated by the application. The required size of this working memory depends on the number of files and directories that can be opened simultaneously. You can get the required working memory size by calling `nn::fs::GetRomRequiredMemorySize`. If this function returns a negative value, you cannot mount a ROM archive. After your application has allocated the required amount of memory (or more) with `nn::fs::WORKING_MEMORY_ALIGNMENT` (4 bytes) alignment, call `nn::fs::MountRom()` to mount the ROM archive.

**Code 7-6. Mounting a ROM Archive**

```
s32 nn::fs::GetRomRequiredMemorySize(size_t maxFile, size_t maxDirectory,
                                     bool useCache = true);
nn::Result nn::fs::MountRom(const char* archiveName, size_t maxFile,
                            size_t maxDirectory, void* workingMemory,
                            size_t workingMemorySize, bool useCache = true);
```

Specify the number of files and directories that can be opened simultaneously in *`maxFile`* and *`maxDirectory`*, respectively. The number of files that can be opened simultaneously only depends on the amount of working memory. This number is not affected by factors like the length of filenames.

Specify a value of `true` in the *`useCache`* parameter to cache metadata and shorten the time required to open files or scan directories. Note that this increases the amount of working memory required.

Specify the name of the archive to mount in the *`archiveName`* parameter. The archive is mounted to `rom:` if you call the overloaded version that omits this parameter.

Pass the working memory and its size to *`workingMemory`* and *`workingMemorySize`*.

**Note:** You do not need to handle errors for these functions. If an error occurs in a function, an

error screen displays but an error is not returned to the application.

### 7.2.1. Specifying Archive Names

Only single-byte alphanumeric characters and some symbols excluding the colon as archive name delimiter may be used to specify archive names. Names are case-sensitive. Names must be specified as at least one character and no more than eight, including the colon delimiter.

Do not use archive names that start with the dollar symbol ($). For information about characters and words that cannot be used in archive, file, and directory names, see the API Reference.

## 7.3. Save Data

For card-based software, the application-specific save data region is located in the backup memory. For downloaded applications, the save data region is located in an archive file on an SD card. The save data region can be accessed by the FS library as an archive. The function and parameters used to access this archive are the same regardless of whether the accessing application is stored on a Nintendo 3DS Card or an SD card.

**Code 7-7. Mounting, Formatting, and Committing Save Data Regions**

```
nn::Result nn::fs::MountSaveData(const char* archiveName = "data:");
nn::Result nn::fs::MountSaveData(const char* archiveName, bit32 uniqueId);
nn::Result nn::fs::MountDemoSaveData(const char* archiveName, bit32 uniqueId,
                                     bit8 demoIndex);
nn::Result nn::fs::FormatSaveData(size_t maxFiles, size_t maxDirectories,
                                  bool isDuplicateAll = false);
nn::Result nn::fs::CommitSaveData(const char* archiveName = "data:");
```

Call the `nn::fs::MountSaveData()` function to mount a save data region to the archive path name specified in the *`archiveName`* parameter. The save data from another application can be mounted by calling the overloaded function with the *`uniqueId`* parameter. Call the `nn::fs::MountDemoSaveData()` function to mount the save data region of the demos. Specify the index of the demo for `demoIndex`.

**Note:** The values to be specified in *`uniqueId`* upon mounting the save data region are based on the unique ID specified in each application's RSF file. To specify another application's unique ID, you must first specify that application's unique ID in the RSF file of the application to be mounted.

If an error belonging to the `nn::fs::ResultNotFormatted`, `nn::fs::ResultBadFormat`, or `nn::fs::ResultVerificationFailed` is returned in the save data region mount, format the save data region by calling the `nn::fs::FormatSaveData()` function and then try mounting it again. When formatting, specify the maximum number of files and directories using the *`maxFiles`* and *`maxDirectories`* parameters. There are no restrictions on the values that can be specified. For more information about specifying archive names, see 7.2.1. Specifying Archive Names.

**Warning:** The same error class (`nn::fs::ResultNotFound`) is returned when mounting the save data region from a downloaded application (including demos) if a Nintendo 3DS Card is not inserted, or if the inserted 3DS Card does not have a formatted save data region.

You cannot escape from this state if you do not start the card-based software application, or start the card-based software application but do not format the save data region. Be sure to display a message that accommodates the reason the caused the error. For example, your message could say something like "Could not find save data for (media name). If you have never run the game before, run it now, create save data, and try again."

**Note:** Depending on the media, some errors are not returned.
For example, currently, CARD1 may return `nn::fs::ResultBadFormat` when it is mounted, but CARD2 and downloaded applications will not return the error `nn::fs::ResultBadFormat`.
We recommend that you write media independent processing that can adapt to future changes in error handling even in such situations.

The library supports automatic redundancy for the entire save data region. The library mirrors data over the entire save data region when the *`isDuplicateAll`* parameter during formatting was specified as `true`. When writing files with automatic redundancy enabled, you must call `nn::fs::CommitSaveData` before unmounting the save data region or the file updates will not be written to the media. Likewise, if the power is cut before the updates are committed to memory, only the old data will be available the next time the system is booted and the save data is mounted.

**Warning:** If save data consists of multiple interdependent files, you must call `nn::fs::CommitSaveData` when these dependencies are not contradictory.

When automatic redundancy is enabled, the memory available for save data files is half of the physical capacity of the backup region. A portion of this space is also reserved for file system management. Use the *Worksheet for Calculating the Save Data Capacity*, found in the API Reference, to calculate the actual available capacity of the backup region. The current implementation uses regions of 512 byte blocks when saving files.

The mounted save data region can be treated as an archive. You can create files freely within the archive. Filenames and directory names can be up to 16 characters long, and the maximum path length is 253 characters. Only half-width alphanumeric characters and certain symbols can be used for filenames and directory names. Slashes ("/") are used to delimit folders in paths. Other than the amount of available capacity for save data, there are no restrictions on the maximum size of the file that can be created or the maximum size that can be written at one time.

Call the `nn::fs::GetArchiveFreeBytes()` function to get the available capacity of an archive.

**Code 7-8. Getting an Archive's Available Capacity**

```
nn::Result nn::fs::GetArchiveFreeBytes(s64* pOut, const char* archiveName);
```

For the *`pOut`* parameter, specify a pointer to a variable of type `s64` to receive the number of available bytes. For the *`archiveName`* parameter, use the name of the archive specified when it was mounted.

The files within the archive are protected by a tamper detection feature that uses hash values. If the

hash value of the data that is accessed does not match the expected value, the system determines that the data has been tampered with and returns an error. When automatic redundancy is not enabled, mismatching hash errors can also occur when trying to access data that was corrupted if the system was turned off during a save or if the card was removed during a save.

When you finish accessing the save data region, call the `nn::fs::Unmount()` function, specifying the archive name as an argument, to unmount the save data.

**Code 7-9. Unmounting**

```
nn::Result nn::fs::Unmount(const char* archiveName);
```

Downloaded applications, much like Nintendo DSiWare titles, create a save data region automatically as soon as they are imported into the target media. When a downloaded application is deleted, its save data is also deleted.

## 7.3.1. Measures Against Rollback of Save Data

**Note:** See the File System: Save Data Rollback Prevention Feature section in the *CTR-SDK API Reference*.

**Warning:** In general, if an application uses the save data rollback prevention feature, do not support backing up save data in the banner specifications file. (Set `DisableSaveDataBackup` to `True`.) For more information, see the *3DS Overview* developer manual, included in the CTR Overview package, and the description of `ctr_makebanner`, included in the `../documents/tools/` folder of the CTR-SDK. Switching between using and not using the save data rollback prevention feature when applying a patch is prohibited.

## 7.3.2. Handling Errors When Files or Directories Do Not Exist

Even if an application takes various measures to create save data correctly, if the following conditions are met, it is still possible to reach a state where mounting is completed successfully, but save data files do not exist.

- The save data was formatted the first time it is created.
- After the formatting is completed, while creating files or directories (and before the save data is committed when automatic redundancy is enabled), the application crashes for one of the following reasons.
  - The Power Button was held down.
  - The card was removed.
  - Power was cut off partway through the process.
    This occurs particularly in applications that allow the system to go into a sleep state, when the system is closed and then left alone.

In cases like these, `nn::fs::ResultNotFound` could be returned for file or directory operations. Implement your application so that the user's save data can be restored to its normal state.

## 7.4. Extended Save Data

The term *extended save data* refers to data that is created on an SD card and managed separately from the save data. The extended save data region can only be used on the system it was created on. We recommend that the extended save data region be used to store any application-specific data that is linked to (relies on) information that only exists on a specific system (for example, friend information). Do not use the extended save data region to store data that is required in order to make progress in the game. The system never automatically creates extended save data.

**Code 7-10. Mounting, Creating, and Deleting Extended Save Data**

```
nn::Result nn::fs::MountExtSaveData(const char* archiveName,
                                    nn::fs::ExtSaveDataId id);
nn::Result nn::fs::CreateExtSaveData(nn::fs::ExtSaveDataId id,
                                     const void* iconData, size_t iconDataSize,
                                     u32 entryDirectory, u32 entryFile);
nn::Result nn::fs::DeleteExtSaveData(nn::fs::ExtSaveDataId id);
```

To use extended save data, you must first mount it by specifying the extended save data ID in a call to the `nn::fs::MountExtSaveData()` function. If the function returns a value of `nn::fs::ResultNotFormatted`, `nn::fs::ResultNotFound`, `nn::fs::ResultBadFormat`, or `nn::fs::ResultVerificationFailed` and extended save data has already been created, delete extended save data using the `nn::fs::DeleteExtSaveData()` function and create extended save data by calling the `nn::fs::CreateExtSaveData()` function and then remount. If an error belonging to `nn::fs::ResultOperationDenied` has been returned, it may be that the SD card or file cannot be written to, or that there is a contact fault in the SD card. See the File System: Error Handling section in the *API Reference*.

The extended save data is mounted to the archive path specified in the *`archiveName`* parameter. For more information about specifying archive names, see 7.2.1. Specifying Archive Names.

Specify the extended save data ID in the *`id`* parameter. The extended save data ID is a number that identifies the extended save data. The extended save data ID for extended save data that applications can access must be set as either `ExtSaveDataNumber` in the makerom RSF file (only one ID can be specified) or `AccessibleSaveDataIds` in `AccessControlInfo` (where multiple IDs can be specified). Normally you would specify a unique ID issued by Nintendo for extended save data IDs created by applications. Consequently, when sharing extended save data among multiple applications, you must specify the unique ID for one of those applications.

When creating extended save data, the *`iconData`* and *`iconDataSize`* parameters are used to specify an icon for display on the data management screen (as an ICN file created using the ctr_makebanner32 tool) and the icon's size. Use the *`entryDirectory`* and *`entryFile`* parameters to specify the number of directories and files to store in the save data

The mounted extended save data region can be treated as an archive. You can create files freely within the archive. However, the size of files cannot be changed after they are created unless the `nn::fs::TryCreateFile()` function is used to create them. Filenames and directory names can be up to 16 characters long, and the maximum path length is 248 characters. Only half-width alphanumeric characters and certain symbols can be used for filenames and directory names. Slashes ("`/`") are used to delimit folders in paths. Archive sizes are variable.

The files within the archive are protected by a tamper detection feature that uses hash values. If the hash value of the data that is accessed does not match the expected value, the system determines that the data has been tampered with and returns an error. Mismatching hash errors call also occur when trying to access data that was corrupted if the system was turned off during while writing data or if the card was removed while writing data. Unlike save data, the library does not support extended save data mirroring.

There are no data protection features, so removing an SD card while a file is writing will very likely result in corrupted extended save data. You must re-create any corrupted extended save data after deleting using the `nn::fs::DeleteExtSaveData()` function.

When you finish accessing the extended save data region, call the `nn::fs::Unmount()` function, specifying the archive name as an argument, to unmount the extended save data.

Deleting downloaded applications has no effect on extended save data. Extended save data can be deleted from the Data Management screen of System Settings.

> **Note:** The application is free to use up to a total of 32 MB of extended save data. If you want to use more than 32 MB, please contact Nintendo.

## 7.4.1. Uses of Extended Save Data

Extended save data can be used to save information such as the following.

- Application-specific data
- Data shared by a series (shared among multiple titles or versions of titles in the same series)
- Downloaded data (such as additional items or levels)
- Contextual CTR banner data (created in the application, or downloaded from a server)

### 7.4.1.1. Application-Specific Data

This can include relatively unimportant data that is unrelated to the game's progress, data that is linked to information that is only saved on the system, or large user-created data.

### 7.4.1.2. Data Shared by a Series

This refers to data that is shared by multiple titles in the same series (including different versions of a title). You can share data between titles in a series by using the common extended save data ID throughout the series.

### 7.4.1.3. Downloaded Data

This refers to additional levels or other such data that are downloaded by the application by registering a download task. Only the application that registered a download task can access the data obtained from that download task, but you can make this data accessible to other applications that share the extended save data by moving it to extended save data.

#### 7.4.1.4. Contextual CTR Banner Data

This refers to data that can replace portions of the data in CTR title banners. The portions that can be replaced are the scrolling text that is displayed at the bottom of the upper screen and a single texture for the replacement name. The system does not support replacing icon data.

There are two types of contextual CTR banner data: local contextual banners, which are created by applications, and downloaded contextual banners, which are downloaded from servers.

##### Local Contextual Banners

Local data can be used to display a message based on the game's progress or to change part of the texture that is displayed in the CTR title banner. The text can be up to 255 characters (either single byte or double byte), and the data size can be up to 128 KB. It is possible to include multiple images within a single texture and to apply these images to multiple models. Prepare "COMMON" data used for all languages in addition to any language-specific data that may be enabled for the target region.

##### Downloaded Contextual Banners

This is provided to a server and can be used to change the display. The text and textures have the same specifications as for local contextual banners. You can specify an expiration date (year, month, and day) for downloaded contextual banners.

##### Displaying Contextual CTR Banner Data

The text displayed for the contextual banner data alternates in the following order: (1) text for downloaded contextual banners, (2) text for local contextual banners. If the texture included within a local contextual banner has the same name as a texture included within a downloaded contextual banner, the system prioritizes the downloaded contextual banner’s texture.

### 7.4.2. Access From Multiple Game Cards

More than one person may share a single 3DS system using a single SD card, with both people owning game cards having the same title. This alone does not represent a problem because save data is written to the backup memory of each separate game card. Note, however, that bugs may arise depending on the configuration of the data saved in extended save data.

Specifically, situations like the following may occur.

- The program may run out of control if the integrity of save data and extended save data is lost.
- If you use fixed filenames for save data, files stored in extended save data created on one game card may overwrite that of the other, regardless of the user’s intention.

- Contrary to the developer's intention, the same number of SD cards is required for multiple game cards to be played on the same 3DS system.

For example, unintentional overwriting of data can be avoided by assigning unique directory and filenames when creating files in extended save data. However, rather than using a name freely chosen by the user, such as a character name, as the basis for naming such files, apply a text string such as one that includes the current time. In addition, when writing information linked with extended save data in save data, note that the extended save data associated with that information may not exist on the SD card.

**Warning:** As for contextual CTR banner data, there is no way to handle situations like this. The downloaded contextual banner data or local contextual banner data last created is enabled, regardless of which game card is inserted.

### 7.4.3. Accessing Multiple Items of Extended Save Data

You can specify multiple unique IDs (up to a maximum of six) for the accessible save data attributes (`AccessibleSaveDataIds` in `AccessControlInfo`) in the RSF file to enable access to the following data.

- Save data with the same unique IDs as the unique IDs specified
- Extended save data with the same extended save data IDs as the unique IDs specified

**Warning:** Individual items of extended save data can be deleted in System Settings. Avoid saving data that needs to be as consistent as extended save data whenever possible.

For titles that use CTR contextual banners, create extended save data that takes the title's unique ID as its extended save data ID in addition to the extended save data for shared access. This is because the files used to display the contextual banner must be in the `ExBanner` directory for extended save data created with the title's unique ID.

**Note:** For information about how to write the RSF file, see the reference for the ctr_makerom CTR-SDK tool or the 3DS Programming Manual: Creating Applications.

## 7.5. Archives That Directly Access SD Cards

Call the `nn::fs::MountSdmcWriteOnly()` function to mount an archive that directly accesses the inserted SD card. Note that the `FileSystemAccess` permission attribute under `AccessControlInfo` in the RSF file must specify – `DirectSdmcWrite`.

**Code 7-11. Mounting an Archive That Directly Accesses an SD Card**

```
nn::Result nn::fs::MountSdmcWriteOnly(const char* archiveName = "sdmcwo:");
```

No files or directories can be read in archives mounted using this function, but written files are not encrypted, so they can be read from a PC. Unlike any extended save data on the same SD card, you can change the size of files after they are created.

There are fewer errors possible when mounting this kind of archive than when mounting extended save data, and you can handle these errors in the same way as for extended save data. However, because reading is prohibited, you cannot directly check for the existence of any file or directory. You must attempt writing once and then handle any `nn::fs::ResultAlreadyExists` errors returned by the write function.

Call the `nn::fs::Unmount()` function and pass the archive name as an argument to unmount an archive.

> **Note:** Applications are restricted from writing depending on the path. For more information about restrictions and cautions, see the File System section of the guidelines.

# 7.6. Media-Specific Cautions

## 7.6.1. Nintendo 3DS Cards

#### Verifying Different Access Speeds

The performance of access to ROM archives on Nintendo 3DS Cards by the FS Library can vary greatly depending on the condition of the media. System updates may also improve performance. If you plan to sell your application on Nintendo 3DS Card, verify the following to make sure that changes in access speed do not cause issues.

- For CARD1 applications:
  - Enable **DebugMode** using the Config tool's **Debug Setting** (with latency emulation enabled) and play through all modes.
  - Set the development card speed to **Fast** and play through all modes of the Release build of the application written to a development card

- For CARD2 applications:
  - Enable **DebugMode** using the Config tool's **Debug Setting** (with latency emulation enabled) and play through all modes.
  - From the **PARTNER-CTR Debugger Card Control** dialog box, in the **Card Emulation Control** tab, set the transfer speed in the emulation memory speed settings to **Fast**, and play through all modes of the application (Release build CCI file) loaded in emulation memory.
  - Set the development card speed to **Fast** and play through all modes of the Release build of the application written to a development card

**Warning:**Make sure to test using the latest version of the PARTNER-CTR Debugger/Writer.

## 7.6.2. SD Memory Cards

Note the following points when you implement an application that accesses an SD card, including extended save data or downloadable application stored in the SD card.

### Cautions for Directly Accessing SD Memory Cards

Archives for accessing SD cards mounted using the `nn::fs::MountSdmc()` function (mounted under `sdmc:/` if not otherwise specified) are provided for debugging purposes only. Note that these archives are not accessible from production versions.

Note that this mount point cannot be accessed in retail versions of games. Note also that SD cards may be swapped out while the system is asleep. To remount an SD card that has been removed, first call `nn::fs::Unmount` to unmount it from the file system, and then call the appropriate mounting function to remount.

### Streaming Playback Cautions

In some situations, SD card file access may become slower, such as when the SD card is almost full, it is significantly fragmented, or it is near the end of its service life. Most functions that access SD cards are affected by this and slow down; the effect is particularly large with functions for creating files (for example, `nn::fs::CreateExtSaveData`).

Note that calling these functions while the system is streaming sound or movies may cause choppy audio or dropped frames, because file access blocks until the functions complete executing.

### "Read Disturb" Measures

SD cards have a limit to their data storage capacity in relation to their read capacity. When trying to read beyond the limit, the data can become degraded in a phenomenon known as *read disturb*. Depending on the area where the read disturb phenomenon occurs, all data on the SD card might become unreadable.

Of particular concern is that **this can occur just by opening a file to be loaded from the SD card**. Some SD cards on the market are not sufficiently resistant to the read disturb phenomenon. The best way to lower the risk of this phenomenon occurring is to reduce, as much as possible, the amount of file opening and closing that occurs on the application side. For example, when multiple accesses are required at different offsets to the same file, perform the accesses all within a small number of open and close processes, rather than opening and closing the file for each access.

In addition, by repeatedly reading the same region on the SD card, data retention becomes unstable. Problems can occur most often in access to ROM archives and downloadable content that has not been rewritten. Even if the application is card-based software, be careful if there is a possibility of selling the card-based software as a downloadable application in the future.

As a measure to retain the data longer when the particular data is repeatedly read from the SD card (such as when repeatedly streaming playback of short waveform data), you can read the data from

the SD card to the buffer in internal memory and use the data on that buffer. Making the buffer size 16 KB or more is effective, because the situation improves with a larger size.

This measure is not a restriction, and you are not required to implement it when memory is limited.

> **Note:** For the developers using the NW4C Sound Library, see the description in the `nw::snd::SoundArchivePlayer` class reference.

# 7.7. Difference Between Save Data and Extended Save Data

When an application goes on sale, you can now select the 3DS Card (CARD2) as the storage media for that application, thereby increasing the storage capacity available for application-specific save data. This allows you to save data that had to be saved as extended save data (due to storage capacity issues) in the past. With this change in specifications, you must carefully distinguish between save data and extended save data based on the type and purpose of the data being saved. This chapter describes the distinguishing characteristics of save data and extended save data and where to save each type of data.

## 7.7.1. Characteristics of Save Data and Extended Save Data

Save data and extended save data have the following distinguishing characteristics.

**Table 7-1. Characteristics of Save Data and Extended Save Data**

| Item | Save Data | Extended Save Data |
|---|---|---|
| Redundancy | Supported | Not supported |
| Backup | Supported. (Downloadable applications only.) Usually enabled. (See below.) | Not supported |
| Rollback prevention | Supported. (Downloadable applications only.) Backup must be disabled if this feature is used. | Not supported |
| Storage media | Card-based applications: Card backup region Downloadable applications: SD card | SD card |
| When storage is allocated | Card-based applications: From the first use. Downloadable applications: At time of installation. | When allocated by the application. |
| Removal | Only from inside the application. | Can be removed from inside an application or on the Data Management screen in System Settings. |
| Data loss | The risk of losing data can be reduced by using the backup and redundancy features. | The risk of losing data cannot be reduced. |
| Lost data recovery | With redundancy: Lost data needs to be recovered only when an entire archive is lost. Without redundancy: Lost data needs to be recovered when an entire archive is lost or when individual files or directories are lost. | Lost data needs to be recovered when an entire archive is lost or when individual files or directories are lost. |

| | | |
|---|---|---|
| Ensuring sufficient storage capacity | Unnecessary. (Save data must fit into a fixed amount of allocated storage.) | Must be performed each time a file is created. |

In some applications, the backup feature has been disabled for the following reasons.

- They are using rollback prevention.
- Inconsistencies may arise between extended save data and save data if only save data is rolled back using the backup feature.
- Restoring save data from backups may be problematic.
  For example, users might repeatedly transfer downloadable content or rare or hidden characters or items to other users over and over again, restoring their save data each time.

## 7.7.2. Policy for Selecting the Storage Location for Data

In general, save important data in save data. Important data may include data that can make further progress in a game impossible if lost. For data that can be lost and the player can still progress in the game, and data where the amount of storage required is variable, save in extended save data. Because extended save data can be deleted on the Data Management screen under System Settings, developers must note that this data may disappear at times that are beyond the control of the application. In addition, developers must also consider that packaged versions of applications available on cards may be played on other consoles, and that an SD card other than the one on which the expanded save data accessed last time is stored may be inserted in a console the next time a particular game is played.

Save the following information in save data to protect a user’s data.

- Data necessary for progress in a game
- Data necessary for consistency with game progress data
  If data necessary for consistency with game progress is saved in extended save data and save data is deleted, initialization may be required or consistency of data may be lost if the backup feature is used. This can massively increase work involved in the debugging process, such as requiring you to roll back save data in each location of the game or having to delete save data and then verify the consistency of data.

We recommend that the following data also be saved in save data.

- Important data required to enjoy the game
  Such data includes screenshots taken when a stage is first cleared or congratulatory messages sent from friends.
- Data that requires consistency to be maintained across multiple files
  Such data includes command data and textures used to replay scenes.
  The chance of losing data at the file level is lower with save data. However, with data linked to a single file and saved, the risk of losing the file is the same whether you use extended save data or save data without redundancy.

We recommend that the following data be saved in extended save data if losing it will not adversely affect game progress.

- Data that can be redownloaded from the network or that can be re-created while playing the game
- Data shared with other applications
  We advise developers to save data shared between games, particularly games played in parallel, in extended save data. Developers must note that using shared data in conjunction with the save data backup feature may lead to dishonest activity such as generating an excessive number of items.
- Data where the number of files or amount of storage required is not fixed
  Such data includes music or stages created by the user or videos recorded by the user.
  With extended save data, you specify the number of files and directories that can be saved, but there is no limit on the size of files. In the case of save data, on the other hand, even though the number of directories and files that can be saved is similarly specified when save data is initialized, save data is unsuitable for saving multiple instances of data having an unfixed size because storage capacity is limited.
- Data imported from the save data of another application
  Although you can access the save data of other applications, incompatibilities can occur depending on the particular combination of titles. For this reason, use extended save data even when transferring data from an earlier version. Save data may be accessed, however, when transferring save data from a downloadable demo or another download-only application.



# 8. Time

The 3DS system can handle two kinds of time: ticks as a unit of time, calculated from the system clock; and the real-time clock (RTC) that comes with a battery backup.

The system provides a timer and alarm feature that uses system ticks to allow applications to measure the passage of time.

In addition to a feature to get the current time from the RTC, there is also a feature to set off an alarm at a specified time and date.

## 8.1. Class for Representing Time

The SDK is designed to use the `nn::fnd::TimeSpan` class as an argument to time functions, in part to avoid any confusion about units. Within this class, time is expressed in nanoseconds using 64-bit integers. To prevent ambiguity in the units, no implicit conversion from integers to this data type is provided. However, 0 can be converted implicitly to this data type.

The `nn::fnd::TimeSpan` class has static member functions (the `From*Seconds()` functions) for generating instances of this class from integer values expressing the time in various time units

(seconds, milliseconds, microseconds, or nanoseconds). It also has member functions (the `Get*Seconds()` functions) that can obtain the time value of the instance (measured in seconds, milliseconds, microseconds, or nanoseconds) as an `s64` type.

You can also perform comparison (==, !=, <, >, <=, ,>=) and arithmetic (+, -, +=, -=) operations between instances of this class.

**Code 8-1. Instance Generation and Get Functions for the Various Time Units**

```
// From*Seconds()
static nn::fnd::TimeSpan FromSeconds(s64 seconds);
static nn::fnd::TimeSpan FromMilliSeconds(s64 milliSeconds);
static nn::fnd::TimeSpan FromMicroSeconds(s64 microSeconds);
static nn::fnd::TimeSpan FromNanoSeconds(s64 nanoSeconds);
// Get*Seconds()
s64 GetSeconds() const;
s64 GetMilliSeconds() const;
s64 GetMicroSeconds() const;
s64 GetNanoSeconds() const;
```

## 8.2. Ticks

A tick is the time it takes for the CPU to go through one clock cycle while the CPU is operating at 268 MHz (roughly 3.73 nanoseconds). The number of ticks in one second is defined by the `nn::os::Tick::TICKS_PER_SECOND` constant.

> **Warning:** When running in extended mode on SNAKE, the tick value remains the same as in standard mode. In other words, one tick is the equivalent to three CPU clock cycles while in extended mode.

Tick values are available using the `nn::os::Tick` class, and the conversion constructors and conversion operators let you convert from this class to the `nn::fnd::TimeSpan` class (which represents actual time), and vice versa.

Two constructors are available: one generates an instance from the tick value represented by an `s64` type, and the other generates an instance from the tick value converted from the time represented by an object of the `nn::fnd::TimeSpan` class.

The conversion operators include one to convert from a tick value to an `s64`-type value and another to convert to an object of the `nn::fnd::TimeSpan` class. You can also perform arithmetic (+, -, +=, -=) operations between instances of the `nn::os::Tick` class.

In addition, you can call the `ToTimeSpan()` function to generate an instance of the `nn::fnd::TimeSpan` class that expresses the time span converted from a tick value.

Call the `GetSystemCurrent()` function to get the time elapsed since the system was started up as a tick value (an instance of the `nn::os::Tick` class).

## 8.3. Timers

The timer feature sends a notification when the specified amount of time has passed. A timer object may be in the signaled or non-signaled state, and it transitions from the non-signaled to the signaled state when the specified time has passed. Timer instances are limited to eight at any one time.

Timer objects are defined by the `nn::os::Timer` class. Generate an instance and then call the `Initialize` or `TryInitialize()` function to initialize. When initializing a timer object, you can choose whether to manually or automatically reset its signaled state. After a manual-reset timer enters the signaled state, all threads waiting for that timer to enter the signaled state are released and the timer remains in the signaled state until it is cleared. After an automatic-reset timer enters the signaled state, of those threads that are waiting for it to enter the signaled state, only the highest-priority thread is released, after which the object resets itself to the non-signaled state.

There are single-use timers that only transition to the signaled state once after the allotted time, and multiple-use periodic timers that transition to the signaled state in cycles of the specified period. Call the `StartOneShot()` function to start a single-use timer. Call the `StartPeriodic()` function to start a periodic timer. You can stop both timer types by calling the `Stop()` function.

Call the `Wait()` function to cause a thread to wait until a timer enters the signaled state. Call the `Signal()` function to put a timer in the signaled state without waiting the specified time. After a manually resetting timer enters the signaled state, it remains that way until you call the `ClearSignal()` function.

Call the `Finalize()` function to explicitly destroy an instance.

For more information about applications that use timers, see 5.3.1.1. Cautions When Shutting Down. Be sure to carry out all proper processing, such as destroying object instances, at this time.

## 8.4. Alarms

An alarm is a feature that calls the registered handler after the specified time has passed. Alarms generate threads internally to call the handler. Consequently, you must call the `nn::os::InitializeAlarmSystem()` function to initialize the alarm system prior to using alarms.

Alarm objects are defined by the `nn::os::Alarm` class. Generate an instance and then call the `Initialize` or `TryInitialize()` function to initialize.

There are single-use alarms that only call the handler once after the allotted time, and multiple-use periodic timers that call the handler in cycles of the specified period. Call the `SetOneShot()` function to set a single-use alarm. Call the `SetPeriodic()` function to set a periodic alarm. You can cancel both alarm types by calling the `Cancel()` function.

Call the `CanSet()` function to check whether alarms can be set. This returns `false` if an alarm is set and the handler has yet to be called. Even after canceling an alarm by calling `Cancel`, this check will not return `true` until the handler has been called once.

Handler types are defined below.

**Code 8-2. Alarm Handler Type Definitions**

```
typedef void(* nn::os::AlarmHandler)(void *param, bool cancelled);
```

Any arguments needed when setting the alarm are passed in the *`param`* parameter. Generally, `false` is passed in the *`cancelled`* parameter, but `true` is passed when the handler has been called after the alarm was canceled with the `Cancel()` function.

Call the `Finalize()` function to explicitly destroy an instance.

> **Warning:** In the current implementation, the alarm system uses two threads internally. As a result, the alarm system becomes unstable if you register many handlers that take a long time to complete.

For more information about applications that use alarms, see 5.3.1.1. Cautions When Shutting Down. Be sure to carry out all proper processing, such as destroying object instances, at this time.

# 8.5. RTC

RTC stands for "real-time clock," the hardware clock included in the system. The clock has its own battery backup and continues keeping time even if the system loses power.

You can only set the clock from the system settings. The clock can be set forward as far as 2049/12/31, but the clock itself can keep time until it reaches 2099/12/31. Because the clock will not reach its upper limit until at least 50 years have passed since it was set, there is no need to check whether the clock has reset itself while an application is running.

This section explains how to use RTC time in an application.

## 8.5.1. Class for Representing Dates and Times

The CTR-SDK system provides the `nn::fnd::DateTime` class for representing dates and times. You can get the current time of the RTC by calling the `nn::fnd::DateTime::GetNow()` function, which returns an instance of this class.

The constructor with arguments allows you to specify milliseconds in addition to the year, month, day, hours, minutes, and seconds. The constructor with no parameters generates an instance representing 2000/01/01 00:00:00.000.

The dates and times that can be handled by this class are in the range from `nn::fnd::DateTime::MIN_DATE_TIME` (1900/01/01 00:00:00.000) to `nn::fnd::DateTime::MAX_DATE_TIME` (2189/12/31 23:59:59.999). All years are calculated in the Gregorian calendar with 2000/01/01 (Sat) as the standard, and the dates and times that are calculated assume that one day is exactly 86,400 seconds in length.

When subtracting one `nn:fnd::DateTime` class instance from another, the date and time difference is returned as an instance of the `nn::fnd::TimeSpan` class. When adding an `nn::fnd::TimeSpan` instance to an `nn::fnd::DateTime` instance, the resulting date and time is returned as an instance of the `nn::fnd::DateTime` class.

The year, month, day, day of the week, hour (in 24-hour notation), minute, second, and millisecond are available as date and time parameters. All of these parameters can be obtained using `Get*()` functions, and all of them except for the day of the week can be replaced using `Replace*()` functions. The `Get*()` function for the day of the week parameter returns an enumerated type (`nn::fnd::Week`), whereas the `Get*()` functions for all other parameters return `s32` values. The `Replace*()` functions return new instances that have the replaced parameters. The parameters of the original instance will not be overwritten.

In the versions of the `GetParameters()` and `FromParameters()` functions that take the `nn::fnd::DateTimeParameters` structure as arguments, you can get or replace all parameters at the same time. In calls to `FromParameters`, the member of the structure that indicates the day of the week are ignored. The result is indeterminate if the date/time parameters are replaced with invalid values. You can check whether a particular set of date/time parameters is valid by calling the `IsValidParameters()` function. In functions that take a structure as an argument, the day of the week is also checked for validity, so you must be careful when setting the parameters for the structures. If the day of the week is unknown, check the validity using an overloaded version of the function that does not take the day of the week as an argument.

The `DateToDays()` function returns a value indicating how many days have passed between the reference date (2000/01/01) and the specified date. The result is indeterminate if an invalid date was specified. To check whether a given date is valid, call the `IsValidDate()` function. To check whether a particular year is a leap year, call the `IsLeapYear()` function, which returns 1 if the specified year is a leap year.

The `DaysToDate()` function returns the number of days that have elapsed since the reference date. The `DaysToWeekday()` function returns the number of days that have elapsed since the reference day of the week.

## 8.5.2. RTC Alarm Feature

The system includes an RTC alarm feature. This feature sends a notification to the running application when the time set for the alarm is reached. Alarm times can be set by the minute. Depending on the clock settings, the alarm notification may be delayed by around one minute. The alarm feature is suitable as an alarm clock or for similar uses, but it is not suitable for precise time-based notification applications, such as an hourglass. For these uses, see 8.3. Timers or 8.4. Alarms.

Use the RTC alarm feature in the PTM library. You must first initialize the library by calling `nn::ptm::Initialize` before you can use it. After you are done using it, call `nn::ptm::Finalize`.

**Code 8-3. Initializing and Finalizing the PTM Library**

```
nn::Result nn::ptm::Initialize();
nn::Result nn::ptm::Finalize();
```

The RTC alarm feature notifies the running application that the clock has reached the set time by setting the specified event to the signaled state.

**Warning:** The application cannot receive notification while in Sleep Mode. When displaying the HOME Menu or a library applet, the thread that started the HOME Menu or library applet remains suspended and the event enters the signaled state. Care must be taken not to implement an RTC alarm in a standby thread because it will not have permission to perform graphics or sound operations.

Use the following functions to specify the event to use and to set the alarm time.

**Code 8-4. RTC Alarm Feature Functions**

```
nn::Result nn::ptm::RegisterAlarmEvent(nn::os::Event &event);
nn::Result nn::ptm::SetRtcAlarm(nn::fnd::DateTime datetime);
nn::Result nn::ptm::GetRtcAlarm(nn::fnd::DateTime *pDatetime);
nn::Result nn::ptm::CancelRtcAlarm();
```

Call `nn::ptm::RegisterAlarmEvent` to specify the event to receive the RTC alarm feature notification. The application must generate and initialize the instance of the `nn::os::Event` class passed in the *`event`* parameter.

Call `nn::ptm::SetRtcAlarm` to set the alarm time. For the *`datetime`* parameter, specify an instance of the `nn::fnd::DateTime` class with the instance's time specified in minutes. This function returns `nn::ptm::ResultOverWriteAlarm` if the alarm time is already set, but it does indeed overwrite the alarm time with the new value. There is no immediate notification if the time set is in the past.

Call `nn::ptm::GetRtcAlarm` to get the alarm's current setting. For the *`pDatetime`* parameter, pass a pointer to an instance of the `nn::fnd::DateTime` class to receive the alarm time. The function returns `nn::ptm::ResultNoAlarm` if the alarm is not set.

Call `nn::ptm::CancelRtcAlarm` to cancel a previously set RTC alarm. The function returns `nn::ptm::ResultNoAlarm` if the alarm is not set.

### 8.5.3. Handling Time Modification Offset Values

The system records the cumulative offset value in seconds when the user changes the clock time in system settings either forward or back. Call `nn::cfg::GetUserTimeOffset` to get this offset value. For more information, see 11.2.8. RTC Modification Offset Value.

When starting an application on the same system, you can compare this offset value with the offset from the last time the application was started to check whether the user has changed the clock in the meantime. You can then use this in your application, but note that, when started on a different system, the application must not impede player progress due to different values.

## 8.6. Mixed Use of Ticks and RTCs Prohibited

Ticks (the `nn::os::Tick` class) and RTCs (the `nn::fnd::DateTime` class) handle time similarly, but how time advances and the accuracy is different. If the same time period is measured by tick and RTC, the results are not necessarily the same. For this reason, values obtained with ticks must not be used in combination with values obtained using an RTC.

Ticks differ on individual systems and vary significantly with temperature. They can have a margin of error up to ±300 seconds over a period of a month.

RTCs also differ on individual systems and vary significantly with temperature. They can have a margin of error up to ±60 seconds over a period of a month. `nn::fnd::DateTime::GetNow` returns an interpolated current time value so its speed of advance can vary by up to ±1 s per hour.

Most of the time that time is handled within libraries, such as for sound, animation or streaming playback, the speed of advancing time is again different from that for ticks and an RTC. Because of this, when an application must synchronize with a performance or sound playback, you must understand the specifications of individual libraries and use speeds that match those specifications.

# 9. Threads

The `nn::os::Thread` class defined in the SDK only defines the basic functions for dealing with threads, such as how to start them, join them, get parameters, and change parameters. Threads used by applications must inherit from this thread class.

## 9.1. Initializing and Starting

After the constructor creates a thread, the thread has yet to be either initialized or started. If you are managing the stack in your application, call the `Start()` function to initialize and start a thread; to have the library automatically manage stack memory, call `StartUsingAutoStack()`. Calling the member functions that start with `Try` (`TryStart()` and `TryStartUsingAutoStack()`) returns an error if the functions fail, such as due to insufficient resources.

If the application manages stack memory, a `GetStackBottom` member function (that returns the bottom of the stack as an `uptr` value) must be defined for the object passed as a stack region to the member functions. Both the `nn::os::StackMemoryBlock` class, which allocates stack memory blocks, and the `nn::os::StackBuffer` class, which can be located within static memory and structures, satisfy this condition and can be safely passed as arguments. Make sure that the application does not allow the stack memory to become invalid (released) until after the threads have terminated.

### 9.1.1. Running Application Threads in the System Core

You can allocate some of the system core's CPU time for applications to run application threads in the system core.

To start a thread in the system core, use the `nn::os::SetApplicationCpuTimeLimit()` function to specify the percentage of CPU time to allocate, and then specify 1 for *`coreNo`*. You must allocate CPU time before you start a thread on the system core; failure to do so results in an error.

**Code 9-1. Functions for Allocating CPU Time on the System Core for an Application**

```
nn::Result nn::os::SetApplicationCpuTimeLimit(s32 limitPercent);
s32        nn::os::GetApplicationCpuTimeLimit();
```

Use the *`limitPercent`* parameter to specify the percentage of the system core's CPU time to allocate to your application. You can specify values in the range from 5 to 30. The specified percentage of CPU time is allocated from the beginning of each 2-millisecond cycle. In other words, specifying a value of 25 causes 0.5 milliseconds (2 * 25 / 100) to be allocated to the application and the remaining 1.5 milliseconds to be allocated to the system.

You can use the `nn::os::GetApplicationCpuTimeLimit()` function to get the current ratio of

CPU allocation. This returns 0 by default because no CPU time is initially allocated to the application.

> **Warning:** After you have allocated CPU time for your application, you cannot restore this setting to 0.
>
> Even if no application threads are running, the system will not use the time allocated to the application. As a result, wireless communication and other system core processes slow down after CPU time is allocated to the application.

### 9.1.2. ManagedThread Class

The `ManagedThread` class adds functionality to the `Thread` class. Basic operation for the two classes is the same, except that the functions for initialization and execution are separated. The following table details the distinction.

**Table 9-1. Functions Added to the ManagedThread Class**

| Added Features | Associated Functions |
|---|---|
| Retrieving information related to the stack | `GetStackBufferBegin`, `GetStackBufferEnd`, `GetStackBufferSize`, `GetStackBottom`, `GetStackSize` |
| Name retention | `GetName`, `SetName` |
| Acceleration of ID retrieval | `GetId`, `GetCurrentId` |
| Retrieving the instance corresponding to the current thread | `GetCurrentThread` |
| Enumuration of the thread | `Enumerate` |
| Searching the thread | `FindById`, `FindByStackAddress` |

When sharing resources with the `Thread` class, the total number of `ManagedThread` class threads is subject to the restriction of the number of threads that can be created. Also, twice the local storage is consumed to use `ManagedThread` when the `nn::os::ManagedThread::InitializeEnvironment()` function is called.

## 9.2. Thread Functions

The operations performed by a thread are implemented in the thread function passed to the member function that initializes and starts the thread. Declare a thread function that takes one or no arguments and does not return a value (a `void` return type).

Some of the `Start()` functions of the `Thread` class can be used by a thread function that does not take any arguments, and there is an overloaded version that allows you to use a template to specify the type of argument that is passed to a thread function. Note that because the arguments passed in are copied to the thread's stack, the template must specify arguments of types that can be copied and the amount of space available in the stack will be reduced by an equivalent amount.

## 9.3. Completion and Destruction

A thread completes when its thread function completes or when it is released from the blocked state by the parent `nn::os::WaitObject` class's `Wait*()` functions. The `Join()` function waits for thread completion unconditionally. If you need more fine-grained control (for example, to wait with a timeout), wait for the thread to exit using the `nn::os::WaitObject::WaitOne()` function and then call `Join`. Follow this same approach if you need to avoid thread blocking due to the `Join()` function.

Call the `IsAlive()` function to check whether a thread is still alive (not yet completed).

Call the `Finalize()` function to destroy an unneeded thread. When doing so, you must use the following procedure.

If the thread was started using the `Start` or `TryStart()` functions, you must explicitly call the `Join()` function before destroying the thread. Do not call `Detach`.

If the thread was started using the `StartUsingAutoStack` or `TryStartUsingAutoStack()` functions, you must explicitly call the `Join` or `Detach()` functions before destroying the thread. After calling `Detach`, you can only call the `Finalize()` function or the destructor for that thread.

## 9.4. Scheduling

Threads are scheduled according to their priority, and you can set the priority of any thread. Thread priorities can be specified as integers between 0 and 31, with 0 indicating the highest priority and 31 the lowest. Standard threads specify the `DEFAULT_THREAD_PRIORITY` of 16.

Call the `Yield()` function to yield execution to other threads of the same priority. This has no effect if there are no threads with the same priority.

Call the `Sleep()` function to put threads into a sleep state for a specified time.

When scheduling occurs as a result of an interrupt to a currently executing thread, the interrupted thread is placed at the top of the thread queue and scheduling is controlled so there is as little thread switching as possible. If you try to create and execute a new thread with a priority lower than or equal to the currently executing thread, thread switching may not occur if an interrupt from a system process prevents immediate scheduling. We recommend using events to wait for thread switching if you need to ensure thread switching.

> **Warning:** Specifying a short time and repeatedly calling `Sleep()` places a heavy load on the system core, and reduces the overall system performance.

## 9.5. Getting and Setting Parameters

Each thread has its own parameters: a thread ID and a priority.

To get a parameter, call the `Get*` member function to get the instance parameter; to get the parameter for the current thread, call the `GetCurrent*` member function. Similarly, call the `Change*`

or `ChangeCurrent*()` functions to set a parameter.

Call the `GetMainThread()` function to get the current thread (main thread) object.

#### Thread ID

Each thread is assigned its own ID as a `bit32` type. You can get these IDs by calling the `GetId` or `GetCurrentId()` functions, but the IDs cannot be changed.

#### Priority

You can set the priority for each thread as an `s32` type. Use the `GetPriority` or `GetCurrentPriority()` functions to get the current priority. Call the `ChangePriority` or `ChangeCurrentPriority()` functions to change the priority.

## 9.6. Thread-Local Storage

Each thread for storing `uptr` types has 16 slots of thread-local storage. You can reserve thread-local storage for use by generating an instance of the `nn::os::ThreadLocalStorage` class, but attempting to reserve more than 16 slots will fail, and the application will be forcibly halted by a `PANIC`.

To set a value in thread-local storage, call the `SetValue()` function; to get a value, call the `GetValue()` function. All thread-local storage slots are set to 0 when a thread is started.

Use the `nn::os::ThreadLocalStorage` class constructor with parameters to register a callback function to be called when ending a thread. The thread-local storage value is passed as an argument when the callback function is called.

## 9.7. Synchronization Objects

Access to non-thread-safe libraries or shared resources must be handled by means of the application synchronizing between threads. The SDK provides synchronization objects such as the `CriticalSection` class.

### 9.7.1. Critical Sections

A `CriticalSection` is a synchronization object used to provide mutual exclusion. Allowing only one thread at a time to enter a `CriticalSection` object effectively prohibits multiple threads from accessing the same resource at the same time. `CriticalSection` objects require more memory than mutexes (described below), but are faster in most cases. There is no upper limit on how many can be created as long as there is memory available.

`CriticalSection` objects are defined by the `nn::os::CriticalSection` class. Generate an

instance and call the `Initialize` or `TryInitialize()` function to initialize, and then call the `Enter` or `TryEnter()` functions to enter the `CriticalSection` and lock its resources. If you call `Enter` to lock a `CriticalSection` object that is already locked, execution of the thread that called the function is blocked until the existing lock is released. These objects allow recursive locks. A lock request from the thread that has the original lock will be nested (the nesting level will be incremented) without the thread being blocked. If you call `TryEnter` instead, the function returns only whether it succeeded in entering the object, and the thread is not blocked.

Call the `Leave()` function to release a lock on a `CriticalSection` object. If this function is called from the locking thread, the nesting level is decremented and the lock is released when the nesting level reaches 0.

Call the `Finalize()` function to explicitly destroy an instance.

Use the `nn::os::CriticalSection::ScopedLock` object to lock a `CriticalSection` object. The lock is automatically released when the object goes out of scope.

#### 9.7.1.1. Thread Priority Inversion

`CriticalSection` objects do not inherit the priority of the thread that generates them. Consequently, if low-priority thread A has a lock on a `CriticalSection` object and then high-priority thread B requests a lock on that object, thread B may be blocked by some thread C, which has a priority higher than A, but lower than B. In other words, B's priority is effectively lowered, with B's and C's relative priority levels being inverted.

### 9.7.2. Mutexes

Mutexes are synchronization objects used for providing mutual exclusion. Much like `CriticalSection` objects, these objects effectively prohibit multiple threads from accessing the same resource at the same time. Unlike `CriticalSection` objects, they implement thread priority inheritance, and there is a limit of 32 on how many can be created. Due to priority inheritance, if a high-priority thread requests a lock on a mutex object that is already locked by a low-priority thread, the locking thread's priority is temporarily increased to match the priority of the thread that requested a lock.

Mutex objects are defined by the `nn::os::Mutex` class. Generate an instance and call the `Initialize` or `TryInitialize()` function to initialize, and then call the `Lock` or `TryLock()` functions to lock a mutex object. If you call `Lock` to lock a mutex object that is already locked, execution of the thread that called the function will be blocked until the existing lock is released. Mutex objects allow recursive locks. Requesting another lock from the same thread causes the new lock request to be nested. If you call `TryLock` instead, the function returns only whether it succeeded in locking the object within the timeout period, and the thread will not be blocked.

Call the `Unlock()` function to release the lock on a locked mutex object. `Unlock` can only be called from the thread that has the current lock. The object will not be unlocked until all recursive locks have been released.

Call the `Finalize()` function to explicitly destroy an instance.

Use the `nn::os::Mutex::ScopedLock` class to lock a mutex object. The lock begins when the `ScopedLock` object is created. The lock is automatically released when the object goes out of scope.

## 9.7.3. Events

Events are simple synchronization objects that send notification that an event has occurred. An event object may be in the signaled or non-signaled state, and transitions from the non-signaled to the signaled state when the specified event has occurred. You can synchronize threads by having them wait for an event to occur. You can only have 32 instances of the `Event` class at any one time.

Event objects are defined by the `nn::os::Event` class. Generate an instance and then call the `Initialize()` function to initialize. During initialization, you can choose whether the event is a manually resetting event or an automatically resetting event. When manually resetting events enter the signaled state, they remain in that state until manually cleared, during which time all threads waiting for that event are released. When automatically resetting events enter the signaled state, only the highest priority thread of all those threads waiting for it to enter the signaled state is released, after which the object resets itself to the non-signaled state.

Call the `Wait()` function to cause threads to wait until an event enters the signaled state. You can specify the length of the timeout period and also check to determine whether an event has occurred during the timeout period. If you set the timeout period to 0, control returns immediately, allowing you to check for event occurrence without execution being blocked.

Call the `Signal()` function to put an event in the signaled state. When a manually resetting event enters the signaled state, it remains in that state until manually cleared by calling the `ClearSignal()` function. If one or more threads are waiting when an automatically resetting event enters the signaled state, only the highest priority thread of all those threads waiting for it to enter the signaled state will be released, after which the object resets itself to the non-signaled state. If no threads are waiting for events, the object remains in the signaled state.

Call the `Finalize()` function to explicitly destroy an instance.

### 9.7.3.1. Light Events

Light events are simple synchronization objects that send flag notifications between threads. They are functionally no different from standard events.

Light events are defined using the `nn::os::LightEvent` class. The `nn::os::LightEvent` class is better than the `nn::os::Event` class in most respects. The only exception is that it cannot wait for multiple synchronization objects like `nn::os::WaitObject::WaitAny()` can. We recommend the use of the `nn::os::LightEvent` class whenever possible. There is no upper limit on how many can be created as long as there is memory available.

You must call the `Initialize()` function on instances that were created using the constructor that has no arguments. During initialization, you can choose whether the event is a manually resetting event or an automatically resetting event. The behavior is the same as the `nn::os::Event` class in either case. You can also specify the type of event using the non-default constructor. There is no need to call the `Initialize()` function on instances that were created using the constructor that takes arguments.

With the exception that the `Wait()` member function of the `nn::os::LightEvent` class does not allow you to specify a timeout, the `Wait()`, `Signal()`, `ClearSignal()`, and `Finalize()` member functions of this class work just like the corresponding functions in the `nn::os::Event` class.

The `nn::os::LightEvent` class adds the following member functions.

You can check the flag using the `TryWait()` member function. This function only returns the

flag's status and does not block execution of the thread. For `LightEvents` that are configured as automatically resetting events, the flag is only cleared (that is, set to `false` to indicate the "non-signaled" state) if the flag had previously been set (set to `true` to indicate the "signaled" state). You can use the `TryWait()` function to specify a timeout time.

The `Pulse()` member function releases threads that are waiting for the flag to be set. For automatically resetting `LightEvent` objects, it only releases the single highest-priority thread and clears the flag. Unlike the `Signal()` member function, it clears the flag even if no threads are waiting. For manually resetting events, it releases all threads that are waiting, and then clears the flag.

## 9.7.4. Semaphores

Semaphores are synchronization objects that have counters. Every time there is a request to acquire a semaphore, the semaphore counter decrements by 1, while the thread that requested the semaphore waits for the counter to be greater than 0. When a semaphore is released, its counter is incremented by 1. After its counter is greater than 0, the semaphore is passed to the highest-priority thread of those threads that are waiting on the semaphore. Semaphores can be used to manage resources by limiting the number of threads that can access those resources at the same time. You can only have eight instances of the `Semaphore` class at any one time.

Semaphore objects are defined by the `nn::os::Semaphore` class. Generate an instance and then call the `Initialize` or `TryInitialize()` function to initialize. When initializing, specify the counter's initial and maximum values.

Call the `Acquire` or `TryAcquire()` functions to acquire a semaphore. If the counter is less than 0 when `Acquire` is called, the calling thread blocks until the semaphore can be obtained. You can specify the length of the timeout period when calling `TryAcquire()` and also check to determine whether the semaphore was obtained during the timeout period. If you set the timeout period to 0, control returns immediately, allowing you to check for semaphore acquisition without execution being blocked.

Call the `Release()` function to release an obtained semaphore. You can specify either 1 or something other than 1 for the counter increment value; this value is used to assign the initial value of instances that were created with an initial value of 0. When releasing a semaphore, call the function with no arguments so that the counter increment is 1.

Call the `Finalize()` function to explicitly destroy an instance.

### 9.7.4.1. Light Semaphores

Light semaphores are synchronization objects that have the same features as standard semaphores. Light semaphores are defined using the `nn::os::LightSemaphore` class. The `nn::os::LightSemaphore` class is better than the `nn::os::Semaphore` class in most respects. The only exception is that it cannot wait for multiple synchronization objects like `nn::os::WaitObject::WaitAny()` can. We recommend the use of `nn::os::LightSemaphore` class whenever possible. There is no upper limit on how many can be created as long as there is memory available.

The difference between light semaphores and standard semaphores is that light semaphores do not define a `TryInitialize` member function. The member functions of the `nn::os::LightSemaphore` class work just like the corresponding functions in the `nn::os::Semaphore` class.

## 9.7.5. Blocking Queues

Blocking queues are synchronization objects used for safely passing messages between threads. Thread execution is blocked when attempting to either insert more elements into a queue than the buffer size allows or extract an element from an empty queue. When a thread is waiting for messages from multiple other threads, blocking queues can be used to pass messages from one to many, or from many to many. These blocking queues are equivalent to the "message queue" feature in the NITRO, TWL, and Revolution SDKs.

Two classes define blocking queues: `nn::os::BlockingQueue` and `nn::os::SafeBlockingQueue`. `nn::os::BlockingQueue` uses `CriticalSection` objects internally for synchronizing between threads, which could lead to deadlocks in situations where priorities might be inverted. If this problem is a possibility, use `nn::os::SafeBlockingQueue`, which uses mutexes to avoid such issues. The member variables and functions are the same for both classes.

Generate an instance and then call the `Initialize` or `TryInitialize()` function to initialize. When initializing, specify an array of type `uptr` and the number of array elements to use as the queue buffer.

Call the `Enqueue` or `TryEnqueue()` functions to add an element (of type `uptr`) to the end of the queue. If the queue is full when `Enqueue` is called, the calling thread is blocked until an element can be added to the queue. When calling `TryEnqueue`, control is returned whether or not an element was successfully added, allowing you to attempt to add an element without execution being blocked.

Call the `Jam` or `TryJam()` functions to insert an element at the beginning of the queue. If the queue is full when `Jam` is called, the calling thread is blocked until an element can be added to the queue. When calling `TryJam`, control is returned regardless of whether an element was successfully added, allowing you to attempt to add an element without execution being blocked.

Call the `Dequeue` or `TryDequeue()` functions to remove an element from the beginning of the queue. If the queue is empty when `Dequeue` is called, the calling thread is blocked until an element can be removed from the queue. When calling `TryDequeue`, control is returned regardless of whether an element was successfully removed, allowing you to attempt to remove an element without execution being blocked.

Call the `GetFront` or `TryGetFront()` functions to get the first element in a queue without removing the element. If the queue is empty when `GetFront` is called, the calling thread is blocked until an element can be obtained from the queue (that is, until an element is added to the queue). When calling `TryGetFront`, control is returned regardless of whether an element was successfully obtained, allowing you to attempt to remove an element without execution being blocked.

Call the `Finalize()` function to explicitly destroy an instance.

## 9.7.6. Light Barriers

Light barriers are synchronization objects that wait for the arrival of multiple threads. Until the number of threads specified during initialization is met, any threads that arrive early are made to wait. However, this class cannot be used to wait for the arrival of M out of N threads (M < N).

Light barriers are defined using the `nn::os::LightBarrier` class. You must call the `Initialize()` function on instances that were created using the constructor that has no arguments. During initialization, you specify how many threads to wait for. You can also specify the number of threads to wait for using the version of the constructor that takes arguments. There is no

need to call the `Initialize()` function on instances that were created using the constructor that takes arguments. There is no upper limit on how many light barriers can be created as long as there is memory available.

The `Await` member function waits for other threads to arrive. Thread execution is blocked until the number of threads specified when this function was called have arrived.

### 9.7.7. Deadlocks

You must make sure to avoid deadlocking when using critical sections, mutexes, and semaphores for mutual exclusion between multiple threads.

For example, assume two threads A and B, which need mutex objects X and Y to access resources. If A locks X and B locks Y, both threads A and B will be permanently blocked. This situation is known as being deadlocked.

One simple method to avoid deadlocks is to have all threads that need to access resources request mutex locks in the same predetermined order.

## 9.8. Upper Limit of Resources

The number of instances that can be generated at the same time for threads, synchronous objects, and timers is restricted.

The number of instances that can be generated is restricted for the following classes.

- `nn::os::Thread` class (see 9.1. Initializing and Starting through 9.6. Thread-Local Storage.)
- `nn::os::Event` class (see 9.7.3. Events.)
- `nn::os::Mutex` class (see 9.7.2. Mutexes.)
- `nn::os::Semaphore` class (see 9.7.4. Semaphores.)
- `nn::os::Timer` class (see 8.3. Timers.)

The following sample shows the member functions that are defined to get that number of resources and the upper limit.

**Code 9-2. Member Functions to Get Number of Resources Used and Upper Limit**

```
static s32 GetCurrentCount();
static s32 GetMaxCount();
```

The `GetCurrentCount()` function returns the number of instances currently being used.

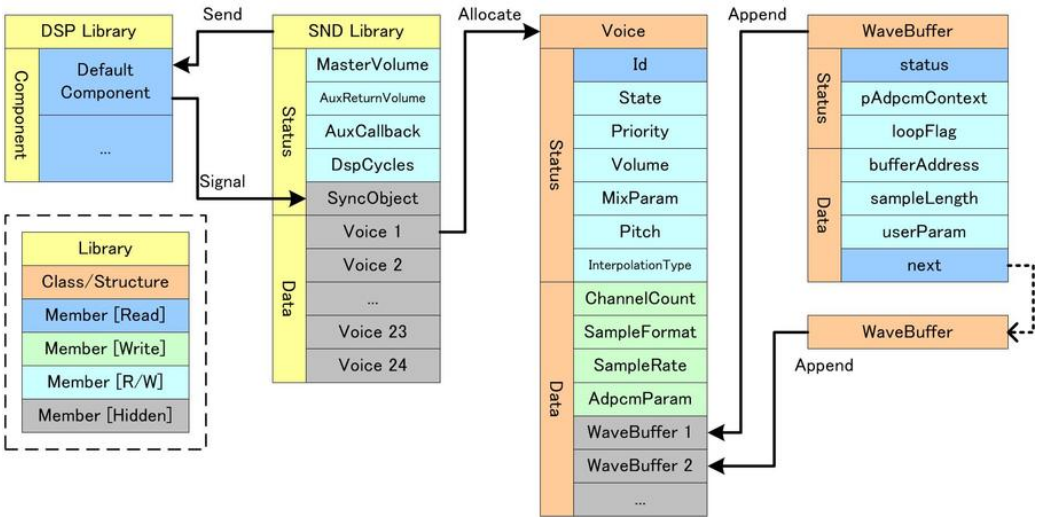
The `GetMaxCount()` function returns the upper limit value for the number of instances that can be generated.

Because the library and other functions begin generating these instances as soon as the application has started, further restrictions to the number of resources the application can use might occur. For more information, see 8. Upper Limit of Resources in the *CTR System Programming Guide*, and the explanations for each library.

# 10. Sound

The CTR-SDK sound library controls sound playback by communicating with the components loaded by the DSP library. The sound library comes with 24 voice objects. An application can allocate the required number of voice objects from this selection and then play back sounds by registering a wave buffer (information about sound source data).

**Figure 10-1. How Sound-Related Libraries Interact**




## 10.1. Initialization

You must use the DSP library to play back sound on the system. Consequently, before initializing the SND library used for sound playback, you must initialize the DSP library and load the components for sound playback.

Use the `nn::dsp::Initialize()` function to initialize the DSP library, and then use the `nn::dsp::LoadDefaultComponent()` function to load the components for sound playback.

Call the `nn::snd::Initialize()` function to initialize the SND library.

**Code 10-1. Initializing the DSP and SND Libraries**

```
nn::Result result;
result = nn::dsp::Initialize();
result = nn::dsp::LoadDefaultComponent();
result = nn::snd::Initialize();
```

### 10.1.1. Setting the Number of Output Buffers

You can use the `nn::snd::SetOutputBufferCount()` function to specify the number of output buffers for the sound data that is ultimately output from the DSP.

**Code 10-2. Setting the Number of Output Buffers**

```
void nn::snd::SetOutputBufferCount(s32 count);
s32 nn::snd::GetOutputBufferCount();
```

You can specify a *`count`* of either 2 or 3. The default is 2.

Two output buffers minimize the delay until sound data is actually played, but sound may break up if there are delays in sound thread processing.

Three output buffers cause sound to be delayed by approximately five milliseconds more than two output buffers, but they also prevent sound from breaking up when there are slight delays in the sound thread.

This function clears the content of the output buffers. Call it immediately after the library is initialized or at another time when sound is not being played.

You can get the current setting with the `nn::snd::GetOutputBufferCount()` function.

## 10.2. Allocating Voice Objects

The SND library synthesizes and then plays sounds on the basis of the information about the sound source data bound to the library's 24 voice objects. In other words, the library can play sounds from up to 24 sound sources at the same time.

Voice objects are limited resources. An application must use the `nn::snd::AllocVoice()` function to allocate voice objects to bind to the sound source data for playback.

**Code 10-3. Allocating Voice Objects**

```
nn::snd::Voice* nn::snd::AllocVoice(
        s32 priority, nn::snd::VoiceDropCallbackFunc callback, uptr userArg);
```

The *`priority`* parameter specifies the priority of the voice object to allocate in the range from 0 to `nn::snd::VOICE_PRIORITY_NODROP` (`0x7FFF` = 32767). The higher the value, the higher the priority. If you specify a value outside of this range, an assert fails and processing halts.

Specifying a value for *`priority`* changes the behavior of the function when the maximum number of voice objects has already been allocated.

When *`priority`* is set to `nn::snd::VOICE_PRIORITY_NODROP` and the lowest-priority allocated voice objects are also `nn::snd::VOICE_PRIORITY_NODROP`, voice object allocation fails and the function returns NULL. For any other value, the lowest-priority allocated voice object is dropped. Attempting to allocate a lowest-priority voice object will fail.

The *`callback`* parameter specifies the callback function to be called after the library forcibly drops a voice object. No callback occurs when this value is NULL. The *`userArg`* parameter specifies any arguments to be passed to the callback. Specify NULL if this is not needed.

The types of the callback functions are defined as follows.

**Code 10-4 Voice Object Drop Callback Function**

```
typedef void (* VoiceDropCallbackFunc)(nn::snd::Voice *pVoice, uptr userArg);
```

The *pVoice* parameter takes a pointer to the dropped voice object, and the *userArg* parameter takes any arguments specified at allocation.

### 10.2.1. Dropped-Voice Mode

The SND library releases (drops) low-priority voice objects when an attempt is made to allocate more than the maximum number of voice objects and when internal DSP processing is in danger of increasing.

Two modes exist to control this latter restriction; you can use the `nn::snd::SetVoiceDropMode()` function to configure them.

**Code 10-5 Setting the Dropped-Voice Mode**

```
void nn::snd::SetVoiceDropMode(nn::snd::VoiceDropMode mode);
```

When *mode* is `VOICE_DROP_MODE_DEFAULT`, only predicted values are used to determine whether to drop voice objects.

When *mode* is `VOICE_DROP_MODE_REAL_TIME`, both predicted values and the actual processing load are used to determine whether to drop voice objects. You must configure three output buffers to be used when you set this mode.

## 10.3. Setting Sound Source Data Information

The memory region (buffer region) for storing sound source data must be allocated as a contiguous 32-byte-aligned region in device memory. The buffer region must be 32-byte aligned and sized to a multiple of 32 bytes. When allocating sound source data buffers from this region, buffers must be contiguous memory regions and the starting addresses must be 32-byte aligned, but the 32-byte multiple size restriction does not apply.

In some cases, sound source data written to a buffer might only be written to the cache. Be sure to call the `nn::snd::FlushDataCache()` function to write the data to memory.

Pass the sound source data to the DSP as an `nn::snd::WaveBuffer` sound source data information structure. Sound source data information must be initialized with the `nn::snd::InitializeWaveBuffer` function before configuring information about the sound source data. This is also the case when reusing information about sound source data that has already been played.

After initializing the sound source data, set the following member variables: `bufferAddress` for the starting address of the buffer, `sampleLength` for the sample length, and `loopFlag` for marking whether to loop playback. Specify the DSP ADPCM sample format using the `pAdpcmContext` member variable. You can specify a value of your own choosing for the user parameter (`userParam`). Do not change the values of the other members of this structure.

Set other information about the sound source data (number of channels, sample format, sampling

rate, and basic information about ADPCM parameters) in the voice object.

Use the `nn::snd::Voice::SetChannelCount()` function to set the number of channels. Set this number to 1 for monaural data, and 2 for stereo data. Any other values are invalid.

Use the `nn::snd::Voice::SetSampleFormat()` function to set the sample format. The SND library supports 8-bit, 16-bit PCM, and DSP ADPCM formats.

**Table 10-1. List of Sample Formats**

| Value | Type | Stereo Playback |
|---|---|---|
| `SAMPLE_FORMAT_PCM8` | 8-bit PCM | Interleaved |
| `SAMPLE_FORMAT_PCM16` | 16-bit PCM | Interleaved |
| `SAMPLE_FORMAT_ADPCM` | DSP ADPCM | No |

When the sample format is DSP ADPCM, set the ADPCM parameters using the `nn::snd::Voice::SetAdpcmParam()` function. Set the `nn::snd::WaveBuffer` member variable `pAdpcmContext` to the address of the ADPCM context data structure stored in the sound source data, and do not change the context data until after playback has finished.

Use the `nn::snd::Voice::SetSampleRate()` function to set the sampling rate. Use the frequency of the sound source data.

Use the `nn::snd::Voice::SetInterpolationType()` function to set the method of interpolating the sound data. Use the `nn::snd::Voice::GetInterpolationType()` function to get the current setting. Three interpolation methods are supported, as shown in the following table. The default value is `INTERPOLATION_TYPE_POLYPHASE`.

**Table 10-2. List of Interpolation Methods**

| Value | Interpolation Method |
|---|---|
| `INTERPOLATION_TYPE_POLYPHASE` | Interpolation using four points. The optimal coefficients are chosen based on the specified sampling rate and pitch. |
| `INTERPOLATION_TYPE_LINEAR` | Linear interpolation. |
| `INTERPOLATION_TYPE_NONE` | No interpolation. Noise occurs in the sound played back when the sound source data sampling frequency is not 32,728 Hz. |

After setting all the information about the sound source, use the `nn::snd::Voice::AppendWaveBuffer()` function to register the sound source to the voice object. Multiple sound sources can be registered consecutively, but 4 is the maximum number of sound sources that can be played back by one voice object in one sound frame (roughly 4.889 ms).

Use the `nn::snd::WaveBuffer` member variable `Status` to check the state of a sound source after registration. This state is `STATUS_FREE` before a voice object is registered, `STATUS_WAIT` while waiting for playback, `STATUS_PLAY` during playback, and `STATUS_DONE` after playback has finished. Sound source data is managed by the voice object when in `STATUS_WAIT` and `STATUS_PLAY`, so do not change any settings when in these states.

You can modify a registered sound source by calling `nn::snd::Voice::UpdateWaveBuffer`. However, the only information that can be modified is the sample length (`sampleLength`) and loop specification flag (`loopFlag`). Additionally, depending on when the data is updated using `SendParameterToDsp`, data after the specified sample length might be played back, or modifications might be invalid because playback is already completed.

You can delete a registered sound source by calling `nn::snd::Voice::DeleteWaveBuffer`. Do not overwrite this data by using `SendParameterToDsp` to update the data until playback is

complete.

For streaming playback of DSP ADPCM sound source data, you only have to set the `pAdpcmContext` member variable of the first sound source data registered to the voice object. The context is not updated for sound source data registered later if their `pAdpcmContext` member variables are set to `NULL`.

## 10.4. Creating Sound Threads

Design your application to create threads (sound threads) and have the threads wait for notifications from the DSP library to the SND library. Notifications occur roughly every 4.889 ms. Set the priority of the created threads as high as possible and avoid any intensive processing within the threads to keep sound playback from being interrupted.

Basically, what sound threads do is loop the following processes.

1. The sound thread waits for notification from the DSP library using the `nn::snd::WaitForDspSync()` function.
2. The application checks the state of the sound source data information registered to the allocated voice object. After playback has finished, it registers the next sound source data information for playback using the `nn::snd::Voice::AppendWaveBuffer()` function. When doing so it normally only registers the sound source data information within the sounds thread, while another thread has already loaded the sound source data to be used for sound playback into a buffer.
3. The sound thread calls the `nn::snd::SendParameterToDsp()` function to send any parameters that have changed since the previous notification and any newly registered sound source data information to the DSP to update sound playback.

**Warning:** Functions that manipulate voice objects are not thread-safe, with the exception of `nn::snd::AllocVoice` and `nn::snd::FreeVoice`. You must ensure mutual exclusion (for example, by using a critical section) when calling these functions by threads other than the sound thread.

## 10.5. Sound Playback

Starting, stopping, and pausing sound playback is carried out by changing the state of a voice object. Use the `nn::snd::Voice::SetState` and `nn::snd::Voice::GetState()` functions to set and get the voice object state. Set the state to `STATE_PLAY` to start playback, `STATE_STOP` to stop playback, and `STATE_PAUSE` to pause playback.

**Table 10-3. Voice Object States**

| State | Description |
|---|---|
| `STATE_PLAY` | Order to start sound playback or an indication that playback is underway. |
| `STATE_STOP` | Order to stop sound playback or an indication that playback is stopped. |

| | |
|---|---|
| STATE_PAUSE | Order to pause sound playback or an indication that playback is paused. Set the state to STATE_PLAY to resume playback. |

State changes may not be applied immediately. There may be a slight lag until parameters are sent by the nn::snd::SendParameterToDsp() function (in other words, until the sound frame update period (roughly 4.889 ms) has passed).

There are various other playback settings aside from states.

### Priority

Use the nn::snd::Voice::SetPriority() function to set the priority of a voice object. Use the nn::snd::Voice::GetPriority function to get the current setting. Sounds are played in the order of voice object priority (for objects of the same priority, the most recently allocated object is played first), so there is a chance that low-priority voice objects might not be played if the processing load is too great.

### Master Volume

The master volume is the overall volume for the SND library. Use the nn::snd::SetMasterVolume() function to set this. Set to a value of 1.0 to play sounds at their normal volume. Call the nn::snd::GetMasterVolume() function to get the current setting.

### Volume

Represents the volume for each voice object. Use the nn::snd::Voice::SetVolume() function to set the volume. Use the nn::snd::Voice::GetVolume() function to get the current setting. Set the volume to 1.0 to specify unity gain.

### Mix Parameters

Mix parameters control the gain for four channels (FrontLeft/FrontRight/RearLeft/RearRight).

Use the nn::snd::Voice::SetMixParam() function to set these parameters. Use the nn::snd::Voice::GetMixParam function to get the current settings. The gain values for the left and right channels are stored in the members of the nn::snd::MixParam structure. Set the gain to 1.0 to play the channel at its normal volume.

The sound playback volume is the product of the master volume, volume, and mix parameter values.

### Pitch

Pitch is the speed applied to the sampling rate of the sound source data. Use the nn::snd::Voice::SetPitch() function to set the pitch. Use the nn::snd::Voice::GetPitch function to get the current setting. Set the pitch to 1.0 to play the data at its normal rate. Setting the pitch to 0.5 for sound source data with a sampling rate of 32 kHz would cause it to play at 16 kHz.

### Checking

Use the nn::snd::Voice::IsPlaying() function to check whether a sound is currently playing.

Use the nn::snd::Voice::GetPlayPosition() function to get the current playback position. The value that is returned indicates the starting position (measured in the number of samples) in main memory of the next value that will be processed by the DSP.

### AUX Buses

The two AUX buses (A and B) can be used to apply effects like delay to the sounds being played. To set and get the volume of the AUX buses, call nn::snd::SetAuxReturnVolume() and nn::snd::GetAuxReturnVolume(), respectively. Set the volume to 1.0 to specify unity gain. Because the mix parameters contain members for setting the gain of the AUX buses, you can adjust the audio data that is sent to the AUX buses for each channel independently.

To apply an effect to sounds on the AUX buses, set a callback function for each bus. Set callbacks using nn::snd::RegisterAuxCallback, and clear callbacks using nn::snd::ClearAuxCallback. To get the callback that is currently set, call nn::snd::GetAuxCallback. The types of the callback functions are defined as follows.

**Code 10-6. Callback Functions for AUX Buses**

```
typedef void(*AuxCallback)(nn::snd::AuxBusData* data, s32 sampleLength,
                           uptr userData);
```

The sampling length (not the size in bytes) is passed into the *sampleLength* parameter, and the sound data is passed into the *data* parameter. Buffer addresses are set for each channel; by overwriting the data in these buffers, you can play sound data with effects applied as the output from the AUX buses.

### Voice Interpolation

You can set the interpolation method used when converting from the sound source data's sampling rate to the desired playback frequency. Call the nn::snd::Voice::SetInterpolationType or nn::snd::Voice::GetInterpolationType() functions to set or get the interpolation method.

You can set the interpolation method to the default four-point (INTERPOLATION_TYPE_POLYPHASE), linear (INTERPOLATION_TYPE_LINEAR), or none (INTERPOLATION_TYPE_NONE). With four-point interpolation, the library chooses the optimal coefficients based on the sampling rate and pitch.

### Filters

You can apply a filter to each voice object. Call the nn::snd::Voice::SetFilterType or nn::snd::Voice::GetFilterType to set or get the filter setting. You can set the filter to monopolar (FILTER_TYPE_MONOPOLE), bipolar (FILTER_TYPE_BIQUAD), or the default (FILTER_TYPE_NONE). There are also functions for setting the coefficients for each filter.

Call the nn::snd::Voice::SetMonoFilterCoefficients or nn::snd::Voice::GetMonoFilterCoefficients() functions to set or get the monopolar filter coefficients. Even if you don't alter the coefficients, you can specify a cutoff frequency to act as a monopolar low-pass filter.

Call the nn::snd::Voice::SetBiquadFilterCoefficients or nn::snd::Voice::SetBiquadFilterCoefficients() functions to set or get the bipolar filter coefficients. This filter type only allows the coefficients to be specified.

When setting using coefficients, calculate the coefficients for each filter using a base frequency of 32,728 Hz.

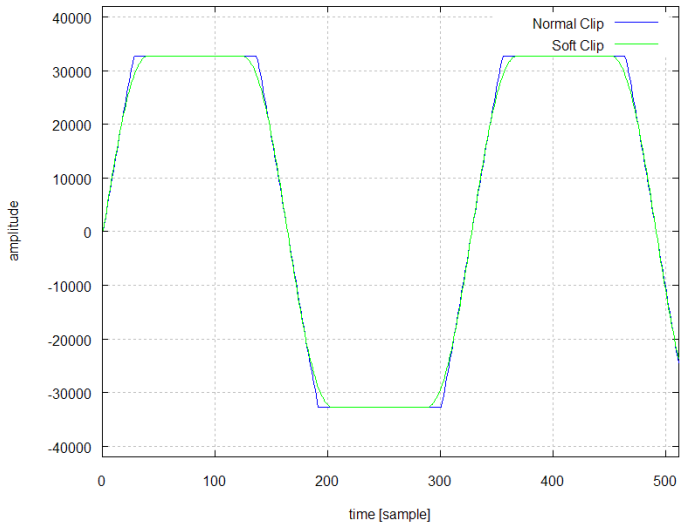
## Clipping Mode

Any portion of the final sound output that exceeds 16 bits is clipped. Two clipping methods are used: either normal clipping or soft clipping that reduces high-frequency noise. Call the `nn::snd::SetClippingMode` to set the clipping mode. Specify either `CLIPPING_MODE_NORMAL` or `CLIPPING_MODE_SOFT`. Call the `nn::snd::GetClippingMode()` function to get the current setting. Soft clipping is the default clipping mode.

The sound output is clipped non-linearly when using soft clipping, which reduces distortion, but this also corrects sample values that are not at the maximum amplitude. This has almost no effect on normal music and most other audio, but harmonics such as single-frequency sine waves may be added for sound input.

The following figure shows the waveforms for a high-amplitude sine wave after clipping in each mode. The green shows the results of soft clipping.

**Figure 10-2. Clipping Mode Differences**




## Automatic Fade-In

You can automatically fade in audio from a volume of 0 at the start of playback to the previously set volume over the course of a single sound frame. You can set fade-in processing for each voice object using the `nn::snd::Voice::SetStartFrameFadeInFlag()` function, passing `true` as the argument to enable fade-in. The default is `false` (no fade-in).

## Using BCWAV Files

Use the `SetupBcwav()` function of the `nn::snd::Voice` class to easily use the waveform files converted by the ctr_WaveConverter tool (BCWAV files) for sound playback.

**Code 10-7. Using BCWAV Files**

```
bool nn::snd::Voice::SetupBcwav(uptr addrBcwav,
  nn::snd::WaveBuffer* pWaveBuffer0, nn::snd::WaveBuffer* pWaveBuffer1,
  nn::snd::Bcwav::ChannelIndex channelIndex = nn::snd::Bcwav::CHANNEL_INDEX_L);
```

For the *addrBcwav* parameter, specify the starting address of a buffer to load the bcwav file to. The

data loaded into the buffer is used (as is) as the sound source data, so the buffer is also subject to the requirements for sound source data. Specifically, the memory must be in device memory with the starting address 32-byte aligned, the buffer size must be a multiple of 32 bytes, and the loaded data must be written to memory using the `nn::snd::FlushDataCache()` function.

For the *`pWaveBuffer0`* and *`pWaveBuffer1`* parameters, specify pointers to the `nn::snd::WaveBuffer` structures used for initial and loop playback. The structures must be created by the application. If you know that the sound source does not have a loop flag set, you can pass `NULL` to the *`pWaveBuffer1`* parameter.

For the *`channelIndex`* parameter, when using stereo sound source data, specify which channel's sound source data to allocate to the `Voice` class. When using monaural sound source data, you must specify `CHANNEL_INDEX_L` for this parameter. In the current version, stereo sound source data converted by the `ctr_WaveConverter` tool is not interleaved. Consequently, you must prepare two instances of the `Voice` class to play stereo BCWAV files.

The function returns `true` if successful and the data is prepared for playback. Of the parameters not included in the BCWAV file header, values of `1.0` are used as the defaults for the volume and pitch. Mix parameters are not set.

Use the `nn::snd::Bcwav` class to access BCWAV file header information.

#### Effects (Delay and Reverb)

You can apply effects to sound playback by either implementing effects in your application by using an AUX bus callback function, or by using the effects provided by the library.

The SND library provides delay (`nn::snd::FxDelay` class) and reverb (`nn::snd::FxReverb` class) effects. To use them, generate an instance of the appropriate class, set the effect parameters, allocate working memory, and then call the `nn::snd::SetEffect()` function, specifying the effect instance and the AUX bus to apply the effect to. You can only apply one type of effect per bus, and if you call `SetEffect` multiple times, only the effect from the last call is applied. The CPU rather than the DSP handles the actual effect processing.

To stop using an effect, call the `nn::snd::ClearEffect()` function, specifying the AUX bus to clear. This function only clears effects, and does not cancel any AUX bus callback function.

> **Note:** For more information about the effect parameters, see the API Reference.

## 10.5.1. Sound Output Mode

The SND library provides three sound output modes: mono, stereo, and surround. Call `nn::snd::SetSoundOutputMode` to set the output mode and `nn::snd::GetSoundOutputMode` to get the current setting.

**Code 10-8. Sound Output Mode Definitions and Getter/Setter Functions**

```
typedef enum
{
    OUTPUT_MODE_MONO       = 0,
    OUTPUT_MODE_STEREO     = 1,
    OUTPUT_MODE_3DSURROUND = 2
} nn::snd::OutputMode;

bool nn::snd::SetSoundOutputMode(nn::snd::OutputMode mode);
nn::snd::OutputMode nn::snd::GetSoundOutputMode(void);
```

If `nn::snd::SetSoundOutputMode` returns `true`, the operation succeeded, and the sound output mode is set to the value specified by the *mode* parameter. By default, the sound output mode is set to stereo (`OUTPUT_MODE_STEREO`).

### 10.5.1.1. Mono

The mix that is output through the left and right speakers is as follows. The four input channels (`FrontLeft`, `FrontRight`, `RearLeft`, and `RearRight`) are the result of mixing the outputs of all voices with the output of the AUX buses.

Output = ( FrontLeft + FrontRight + RearLeft + RearRight ) * 0.5

### 10.5.1.2. Stereo

The mixes that are output through the left and right speakers are as follows. The four input channels (`FrontLeft`, `FrontRight`, `RearLeft`, and `RearRight`) are the result of mixing the outputs of all voices with the output of the AUX buses.

OutputLeft = ( FrontLeft + RearLeft )

OutputRight = ( FrontRight + RearRight )

### 10.5.1.3. Surround

The four input channels (`FrontLeft`, `FrontRight`, `RearLeft`, and `RearRight`) undergo 3D surround processing to impart spatial widening to the output. The four input channels are the result of mixing the outputs of all voices with the output of the AUX buses. This 3D surround operation calculates the speaker output based on the "position" and "depth" parameters of the virtual speakers. Although the 3D surround operation is performed on the mixed output (as opposed to individual voices), it is possible to bypass the output of the front channels (`FrontLeft` and `FrontRight`) on a per-voice basis so that they are not affected by the 3D surround operation.

After 3D surround processing, sound volume tends to seem greater when the pan position is set on either left or right edges compared to when set in the center. If this difference in volume is a concern, correct output accordingly to mitigate the effect, such as by limiting the left and right edge mix parameters to a maximum of roughly 0.8.

#### Positions of the Virtual Speakers

Use `nn::snd::SetSurroundSpeakerPosition` to specify the virtual speaker position mode. If this function succeeds in setting the mode, it returns `true`.

**Code 10-9. Setting the Virtual Speaker Position**

```
bool nn::snd::SetSurroundSpeakerPosition(nn::snd::SurroundSpeakerPosition pos);
```

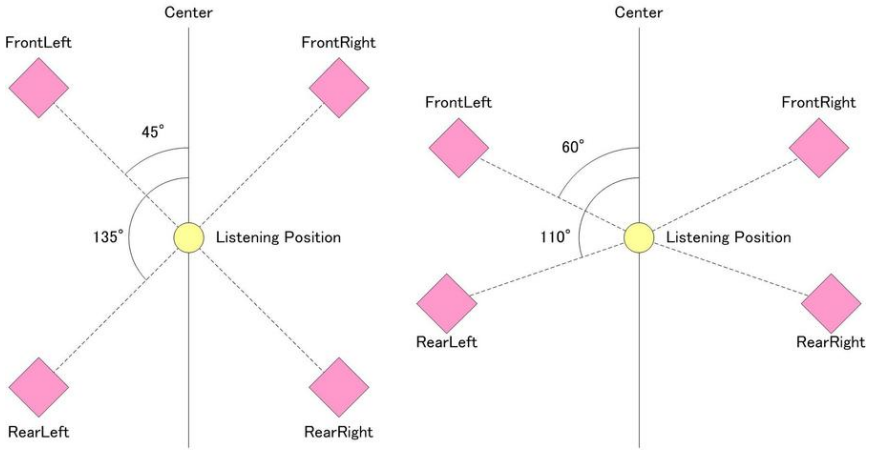
Specify one of the following modes for the *pos* parameter.

**Table 10-4. Modes for the Virtual Speaker Position**

| Setting Value | Description |
|---|---|
| SURROUND_SPEAKER_POSITION_SQUARE | Square mode. Positions the virtual speakers in a square pattern. |
| SURROUND_SPEAKER_POSITION_WIDE | Wide mode. Positions the left and right virtual speakers further apart. |

The virtual speaker mode determines the angle of symmetry of the four channels. The following figure shows the angles at which the speakers are positioned for each mode.

**Figure 10-3. Virtual Speaker Placement for Each Mode**




SURROUND_SPEAKER_POSITION_SQUARE SURROUND_SPEAKER_POSITION_WIDE

## Surround Depth

You can vary the intensity of the surround effect by specifying a depth value using `nn::snd::SetSurroundDepth`. If this function succeeds in setting the depth value, it returns `true`.

**Code 10-10. Setting the Surround Depth Value**

```
bool nn::snd::SetSurroundDepth(f32 depth);
```

Set the *depth* parameter to a value between `0.0` and `1.0`, inclusive. Specifying `0.0` produces the minimum effect, and specifying `1.0` produces the maximum effect. The default value is currently `1.0`.

The 3D surround operation automatically adapts depending on whether the sound is being output through the speakers or through headphones. This depth value is only valid when sound is output through the speakers. It is disabled when sound is output through the headphones and has no effect on the surround effect.

## Front Bypass Setting

The output for the rear channels is always affected by the 3D surround operation. In contrast,

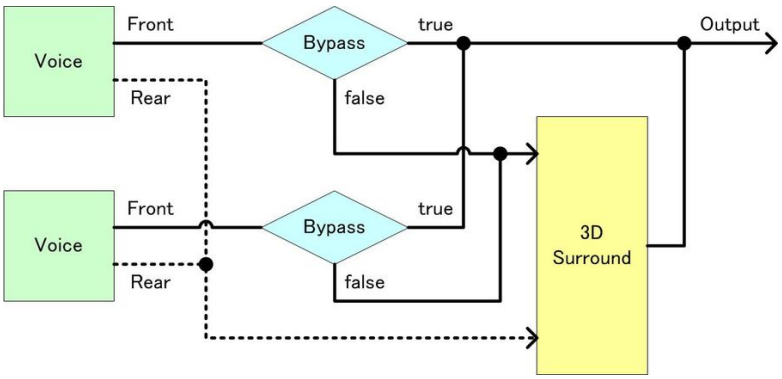
each voice object that is output to the front channels can be configured on an individual basis to bypass the 3D surround effect. You can configure the bypass setting by calling the `SetFrontBypassFlag` member function of the `nn::snd::Voice` class.

**Code 10-11. Front Bypass Setting for Voice Objects**

```
void nn::snd::Voice::SetFrontBypassFlag(bool flag);
```

Set the *`flag`* parameter to `true` to make the output for the front channels bypass the 3D sound operation. This argument defaults to `false` (in which case, the data for the front channels does not bypass the 3D surround operation).

**Figure 10-4. Effect of the Front Bypass Setting on the 3D Surround Operation**




You can also set the front bypass for the front channel of the AUX bus.

**Code 10-12. Setting the AUX Bus Front Bypass**

```
bool nn::snd::SetAuxFrontBypass(nn::snd::AuxBusId busId, bool flag);
```

Specify the AUX bus ID in the *`busId`* parameter, and pass `true` in the *`flag`* parameter to bypass. The default value for all buses is `false` (no bypass).

## 10.5.2. Headphone Connection State

Call the `nn::snd::GetHeadphoneStatus` or `nn::snd::UpdateHeadphoneStatus()` function to find out whether headphones are connected to the audio jack. The function returns `true` when headphones are connected.

The `GetHeadphoneStatus()` function returns the headphone state from the states regularly updated by the `SendParameterToDsp()` function called by the sound thread, and so the results may be up to 32 sound frames (approximately 160 ms) old. Use the `UpdateHeadphoneStatus()` function if you need real-time results. Note that this function entails a heavier processing load, as it updates the DSP state.

**Code 10-13. Headphone Connection State**

```
bool nn::snd::GetHeadphoneStatus();
bool nn::snd::UpdateHeadphoneStatus();
```

## 10.5.3. Getting the Output Audio Data

You can load the mixed audio data that will ultimately be played back through the speakers into a buffer.

**Code 10-14. Getting the Output Audio Data**

```
bool nn::snd::GetMixedBusData(s16* pData,
                              s32 nSamplesPerFrame = NN_SND_SAMPLES_PER_FRAME);
```

For the *pData* parameter, specify a buffer for storing the audio data. The buffer's starting address must be `nn::snd::MIXED_BUS_DATA_ALIGNMENT` (4 bytes) aligned. Calculate the required size of the buffer as `sizeof(s16)` ∗ *nSamplesPerFrame* ∗ 2.

For the *nSamplesPerFrame* parameter, specify the number of samples per channel (usually `NN_SND_SAMPLES_PER_FRAME`).

The function returns `true` if it successfully loads the audio data. Audio data is stored in stereo 16-bit PCM format, with the left and right channels interleaved. Header information is not included. The sampling rate is the sound DSP sampling rate (approximately 32,728 Hz).

## 10.5.4. Encoding to DSP ADPCM Format

Call the `nn::snd::EncodeAdpcmData()` function to encode monaural 16-bit PCM data in the DSP ADPCM format.

**Code 10-15. Encoding to DSP ADPCM Format**

```
s32 nn::snd::GetAdpcmOutputBufferSize(s32 nSamples);
void nn::snd::EncodeAdpcmData(s16* pInput, u8* pOutput, s32 nSamples,
                              s32 sampleRate, s32 loopStart, s32 loopEnd,
                              nn::snd::DspsndAdpcmHeader* pInfo);
```

For the *pInput* parameter, specify the starting address of a buffer storing the monaural 16-bit PCM data to encode. The function can only encode monaural audio data. Specify the number of samples and the sampling rate in the *nSamples* and *sampleRate* parameters.

For the *pOutput* parameter, specify a buffer for storing the encoded DSP ADPCM data. The buffer's starting address must be 4-byte aligned. Get the required size of the buffer by calling the `nn::snd::GetAdpcmOutputBufferSize()` function, passing the number of audio data samples as an argument.

For the *loopStart* and *loopEnd* parameters, specify the starting and ending sample positions for looping. Specify sample position values relative to the start of the 16-bit PCM data, which is considered as `0`.

For the *pInfo* parameter, specify a pointer to a structure storing the DSP ADPCM data header information.

## 10.5.5. Decoding From DSP ADPCM Format

Call the `nn::snd::DecodeAdpcmData()` function to decode audio data in the DSP ADPCM format to monaural 16-bit CPM data.

**Code 10-16. Decoding From DSP ADPCM Format**

```
void nn::snd::DecodeAdpcmData(const u8* pInput, s16* pOutput,
                              const nn::snd::AdpcmParam& param,
                              nn::snd::AdpcmContext& context, s32 nSamples);
```

For the *`pInput`* parameter, specify the starting address of a buffer storing the DSP ADPCM data to decode. For the *`param`* and *`context`* parameters, specify the DSP ADPCM data parameters and context. For the *`nSamples`* parameter, specify the number of audio data samples.

For the *`pOutput`* parameter, specify a buffer for storing the decoded monaural 16-bit PCM data. The buffer's starting address must be 4-byte aligned.

After execution completes, *`context`* stores the context as of the end of decoding.

## 10.5.6. Converting Between Sample Positions and Nibbles

The library provides functions for converting between PCM data sample positions and the number of DSP ADPCM nibbles.

**Code 10-17. Converting Between Sample Positions and Nibbles**

```
u32 nn::snd::ConvertAdpcmPos2Nib(u32 nPos);
u32 nn::snd::ConvertAdpcmNib2Pos(u32 nNib);
```

## 10.5.7. Sound Output When the System Is Closed and Sleep Is Rejected

If sleep is rejected when the system is closed, sound is forcibly output to the headphones, but you can control that behavior with the following function.

**Code 10-18. Controlling Sound Output When the System Is Closed and Sleep Is Rejected**

```
nn::Result nn::snd::SetHeadphoneOutOnShellClose(bool forceout);
```

Specify `false` for the *`forceout`* argument to have sound output from the speakers even when the system is closed (if the headphones are not connected). Specify `true` to have sound output from the headphones regardless of whether the headphones are connected.

**Warning:** If you specify `false` for the *`forceout`* argument, make sure to reject the request to sleep. If the request is not rejected, sound is output from the speakers in the time between when the system is closed and when it transitions to the sleep state.

If this function is called from within the callback function that receives the sleep request, however, that particular sleep request does not change the setting, and the setting takes effect starting with the next sleep request.

## 10.5.8. Source of Noise and Countermeasures

The device design may cause noise in the following circumstances.

### 10.5.8.1. A buzzing noise comes from the speaker

The speaker and physical structure of CTR has shown a tendency for clipping noise to occur around 500 Hz. This noise is caused by the surround process amplifying the playback when playing a sound containing the 500-Hz range with a pan.

It occurs under the following conditions.

- The speaker is playing sound.
- The playback is a sine wave in the 400-Hz to 600-Hz range, or a sine wave like sound, such as a flute or French horn.
- The volume is set to a high value.
- The pan is set to the maximum value.
- Surround is ON.

Possible ways of handling this include:

- Do not pan across extremes.
- Center the playback, and maintain the volume by playing back in moderate volume from the two speakers.
- Suppress the 400-Hz to 600-Hz range using the equalizer.
- Change the tone (waveform) of the playback.

### 10.5.8.2. A mechanical buzzing noise comes from the speaker

This noise, which is caused by the vibration from loud playback, is transmitted to the 3D depth slider. The 3D depth slider and the body rattle against one another other.

This noise becomes most noticeable with a simple sine wave tone in the 400-Hz to 1-kHz range. This noise is conclusively identified by holding down the 3D depth slider with a finger. The noise is this type if the buzzing is muted by the pressure on the slider.

This noise is mitigated by reducing the speaker output volume, or playing a sound that is not a simple sine wave tone.

### 10.5.8.3. Light static sound is superimposed on the playback

This noise is caused by the design of the speaker.

This noise is somewhat prominent in single tones using sine waves at about 400 Hz to 1 kHz.

This noise is mitigated by reducing the speaker output volume, or playing a sound that is not a simple sine wave tone.

### 10.5.8.4. A snapping or popping noise is heard from the earphone under certain circumstances

In the following cases, a snapping or popping noise is audible from the earphones.

- When power is turned on or off
- When entering or leaving Sleep Mode
- When entering compatibility mode

This noise is due to the design of the system. There is no mitigation measure. If the noise is heard when the power is turned on with the volume slider set to zero, the sound always occurs on that unit in these circumstances.

There are production and development units that cause this noise. But this noise does not occur with PARTNER-CTR.

## 10.6. Reusing Sound Source Data Information

Information for sound source data that has been played back can be reused by initializing with the `nn::snd::InitializeWaveBuffer()` function. Use the `status` member of the sound source data information to check whether sound source data playback has finished.

**Table 10-5 Sound Source Data Information States**

| State | Description |
|---|---|
| `STATUS_FREE` | State immediately after initializing sound source data information. |
| `STATUS_WAIT` | State immediately after registering sound source data information to a voice object. |
| `STATUS_PLAY` | State when sound source data is being played. |
| `STATUS_DONE` | State when sound source data playback is finished. Includes when sound playback has been stopped. |
| `STATUS_TO_BE_DELETED` | State when registered sound source data is scheduled for deletion using `nn::snd::Voice::DeleteWaveBuffer`.<br>The state transitions to `STATUS_DONE` the next time `nn::snd::SendParameterToDsp` is executed. |

The following diagram shows these state transitions.

**Figure 10-5. Sound Source Data Information State Transitions**

## 10.7. Releasing Voice Objects

Use the `nn::snd::FreeVoice()` function to release voice objects that are no longer needed. However, voice objects targeted for dropping by the library at the time of allocation must not be released using this function.

## 10.8. Finalizing

Complete the following steps to finalize the DSP and SND libraries used for sound playback.

1. Call the `nn::snd::Finalize()` function to close the SND library.
2. Call the `nn::dsp::UnloadComponent()` function to unload the components loaded into the DSP library and stop the library.
3. Call the `nn::dsp::Finalize()` function to close the DSP library.

**Code 10-19. Finalizing Sound Processing**

```
nn::snd::Finalize();
nn::dsp::UnloadComponent();
nn::dsp::Finalize();
```



# 11. System Settings

This chapter describes how applications can get settings, such as user information and sound settings, from the system settings.

# 11.1. Initialization

Use the CFG library to access information handled by System Settings or about the 3DS system itself.

You must call the `nn::cfg::Initialize()` function to initialize the CFG library before you can use its functionality, with the exception of a few functions not bound by this restriction. After initialization, CFG library functions can be called up until finalization occurs.

**Code 11-1. Initializing the CFG Library**

```
void nn::cfg::Initialize(void);
```

# 11.2. Getting Information

After initializing the library, call the provided functions to access various kinds of information.

## 11.2.1. User Name

You can get the user name configured in the system settings.

**Code 11-2. Getting the User Name**

```
void nn::cfg::GetUserName(nn::cfg::UserName* pUserName);

struct nn::cfg::UserName
{
    wchar_t userName[CFG_USER_NAME_LENGTH];
    bool isNgUserName;
    NN_PADDING1;
};
```

For the *pUserName* parameter, specify a pointer to an `nn::cfg::UserName` structure for storing the user name.

The user name is stored as a wide-character string in the `userName` member of the `nn::cfg::UserName` structure, and `true` is stored in the `isNgUserName` member if the user's name includes any words that fail a profanity check.

In the Japan region, the profanity check uses Japanese, in the North American region it uses American English and the System Settings language, and in Europe it uses British English and the System Settings language.

## 11.2.2. Birthday

You can get the user's birthday configured in the system settings.

**Code 11-3. Getting the User's Birthday**

```
void nn::cfg::GetBirthday(nn::cfg::Birthday* pBirthday);

struct nn::cfg::Birthday
{
    s8  month;
    s8  day;
};
```

For the *pBirthday* parameter, specify a pointer to an `nn::cfg:Birthday` structure for storing the birthday.

## 11.2.3. Country Code

You can get the country code for the user's country and region of residence configured in the system settings.

**Code 11-4. Getting the Country Code**

```
nn::cfg::CfgCountryCode nn::cfg::GetCountry(void);
```

For more information about the country codes defined in the system, see the `nn/cfg/CTR/cfg_CountryCode.h` header file.

Use the following function to make conversions between country codes and country name codes (ISO 3166-1 alpha-2 format).

**Code 11-5. Converting Between Country Codes and Country Name Codes**

```
nn::Result nn::cfg::ConvertCountryCodeToIso3166a2(
    char* iso3166a2, nn::cfg::CfgCountryCode countryCode);
nn::Result nn::cfg::ConvertIso3166a2ToCountryCode(
    nn::cfg::CfgCountryCode* pCountryCode, const char* iso3166a2);
```

**Note:** The `nn::cfg::GetCountryCodeA2()` function will be removed from CTR-SDK.

**Warning:** Applications that use the function for converting between country codes and country name codes must be sure to handle the `nn::cfg::ResultNotFound` return value if it is returned.

## 11.2.4. Language Code

You can get the language code for the language used for display configured in the system settings.

**Code 11-6. Getting the Language Code**

```
nn::cfg::CfgLanguageCode nn::cfg::GetLanguage(void);
```

The following table shows the language codes defined in the system.

**Table 11-1. Language Codes**

| Values | Language | Language Name (ISO 639-1 alpha-2) |
|---|---|---|
| CFG_LANGUAGE_JAPANESE | Japanese | ja |
| CFG_LANGUAGE_ENGLISH | English | en |
| CFG_LANGUAGE_FRENCH | French | fr |
| CFG_LANGUAGE_GERMAN | German | de |
| CFG_LANGUAGE_ITALIAN | Italian | it |
| CFG_LANGUAGE_SPANISH | Spanish | es |
| CFG_LANGUAGE_SIMP_CHINESE | Chinese (Simplified) | zh |
| CFG_LANGUAGE_KOREAN | Korean | ko |
| CFG_LANGUAGE_DUTCH | Dutch | nl |
| CFG_LANGUAGE_PORTUGUESE | Portuguese | pt |
| CFG_LANGUAGE_RUSSIAN | Russian | ru |
| CFG_LANGUAGE_TRAD_CHINESE | Chinese (Traditional) | zh |

Use the following function to covert the obtained language code to a language name in ISO 639-1 alpha-2 format.

**Code 11-7. Converting From Language Code to Language Name**

```
const char* nn::cfg::GetLanguageCodeA2(CfgLanguageCode cfgLanguageCode);
```

The function returns NULL if there is no string corresponding to the language code specified in the *cfgLanguageCode* parameter.

> **Note:** This function can be called without initialization occurring beforehand.

## 11.2.5. Simple Address Information

You can get the user's simple address information (country, region, latitude, longitude) configured in System Settings.

**Code 11-8. Getting Simple Address Information**

```
void nn::cfg::GetSimpleAddress(nn::cfg::SimpleAddress* pSimpleAddress);

struct nn::cfg::SimpleAddress
{
```

```
    u32 id;
    wchar_t countryName[CFG_SIMPLE_ADDRESS_NUM_LANGUAGES]
                       [CFG_SIMPLE_ADDRESS_NAME_LENGTH];
    wchar_t regionName[CFG_SIMPLE_ADDRESS_NUM_LANGUAGES]
                      [CFG_SIMPLE_ADDRESS_NAME_LENGTH];
    u16 latitude;
    u16 longitude;
};
```

For the *pSimpleAddress* parameter, specify a pointer to an nn::cfg::SimpleAddress structure for storing the simple address information.

The countryName and regionName members of the nn::cfg::SimpleAddress structure store the country name and region name as wide-character strings, and the latitude and longitude members store the latitude and longitude.

The latitude and longitude member values are displayed in increments of 360° ÷ 65546 (approximately 0.005°). Northern latitudes from 0° through 90° are stored as values from 0x0000 through 0x4000, southern latitudes from 0.005° through 90° as values from 0xFFFF through 0xC000, eastern longitudes from 0° through 179.995° as values from 0x0000 through 0x7FFF, and western longitudes from 0.005° through 180° as values from 0xFFFF through 0x8000.

## 11.2.5.1. Simple Address Information ID

The following function gets only the simple address information ID.

**Code 11-9. Getting Simple Address Information ID**

```
void nn::cfg::GetSimpleAddressId(nn::cfg::SimpleAddressId* pSimpleAddressId);

struct nn::cfg::SimpleAddressId
{
    u32 id;

    nn::cfg::CfgCountryCode GetCountryCode(void) const;
    u8 GetRegionCode(void) const;
};
```

For the *pSimpleAddressId* parameter, specify a pointer to an nn::cfg::SimpleAddressId structure.

The nn::cfg::SimpleAddressId structure has a GetCountryCode member function to get the country code and a GetRegionCode member function to get the region code. For the values available from these respective member functions, see 11.2.3. Country Code.

## 11.2.5.2. Getting the Simple Address Information From the ID

The simple address information can be obtained from the simple address information ID

**Code 11-10. Getting Simple Address Information From the ID**

```
nn::Result nn::cfg::GetSimpleAddress(nn::cfg::SimpleAddress* pSimpleAddress,
                                     nn::cfg::SimpleAddressId simpleAddressId,
                                     uptr pWorkMemory, u32 workMemorySize);
```

The simple address information obtained based on the ID specified in `simpleAddressId` is stored in *`pSimpleAddress`*.

Working memory is needed for operation of this function. A memory region at least as large as `nn::cfg::CFG_SIMPLE_ADDRESS_WORKMEMORY_SIZE` must be reserved and the memory location and size specified in *`pWorkMemory`* and *`workMemorySize`* respectively.

### 11.2.5.3. Converting Simple Address Information IDs Between 3DS and Wii U

Use the following function to convert simple address information IDs between 3DS and Wii U.

**Code 11-11. Converting Simple Address IDs Between 3DS and Wii U**

```
nn::cfg::SimpleAddressId nn::cfg::ConvertToWiiUSimpleAddressId(
    nn::cfg::SimpleAddressId ctrSimpleAddressId);
nn::cfg::SimpleAddressId nn::cfg::ConvertToCtrSimpleAddressId(
    nn::cfg::SimpleAddressId wiiUSimpleAddressId);
```

## 11.2.6. Region Codes

**Note:** The region codes here are those set in the 3DS system at the time of shipment. Region codes set in Game Cards are specified in a BSF file, and applications will not run if the system and card region codes do not match or if the card does not have a set region code. For more information about BSF files, see the reference manual for the CTR-SDK tool `ctr_makebanner`.

You can get the region code for the system's target market.

**Code 11-12. Getting the Region Code**

```
nn::cfg::CfgRegionCode nn::cfg::GetResion(void);
```

The following table shows the region codes defined in the system.

**Table 11-2. Region Codes**

| Values | Target Market | 3-Letter Code |
|---|---|---|
| CFG_REGION_JAPAN | Japan | JPN |
| CFG_REGION_AMERICA | Americas | USA |
| CFG_REGION_EUROPE | Europe and Australia | EUR |
| CFG_REGION_CHINA | China | CHN |
| CFG_REGION_KOREA | South Korea | KOR |
| CFG_REGION_TAIWAN | Taiwan | TWN |

Use the following function to covert the obtained region code to a three-letter string.

**Code 11-13. Converting From Region Code to 3-Letter String**

```
const char* nn::cfg::GetRegionCodeA3(CfgRegionCode cfgRegionCode);
```

The function returns NULL if there is no string corresponding to the language code specified in the *cfgRegionCode* parameter.

## 11.2.7. Sound Output Mode

You can get the sound output mode configured in System Settings.

**Code 11-14. Getting the Sound Output Mode**

```
nn::cfg::CfgSoundOutputMode nn::cfg::GetSoundOutputMode(void);
```

The following table shows the sound output modes defined in the system.

**Table 11-3. Sound Output Modes**

| Values | Sound Output Mode |
|---|---|
| CFG_SOUND_OUTPUT_MODE_MONO | Monaural |
| CFG_SOUND_OUTPUT_MODE_STEREO | Stereo |
| CFG_SOUND_OUTPUT_MODE_SURROUND | 3D Surround Sound |

## 11.2.8. RTC Modification Offset Value

You can get the offset value saved to the hardware as the cumulative total of user changes to the RTC time.

**Code 11-15. Getting the RTC Modification Offset Value**

```
nn::fnd::TimeSpan nn::cfg::GetUserTimeOffset(void);
```

The function returns the modification offset in seconds. This is the cumulative total of the absolute values of all user changes made to the clock with the system settings. For more information about how to handle these values, see 8.5.3. Handling Time Modification Offset Values.

## 11.2.9. Parental Controls

An application must check whether restrictions are enabled in Parental Controls before photographs and images can be exchanged or friends added. Call `nn::cfg::IsParentalControlEnabled` to check whether the parental controls are enabled.

**Code 11-16. Checking Whether Parental Controls Are Enabled**

```
bool nn::cfg::IsParentalControlEnabled(void);
```

**Table 11-4. Parental Controls Settings and Restricted Functionality**

| Item | Restricted Application Functionality | Section |
|---|---|---|
| Age Restriction | N/A (restricted to the system side) | - |
| Use of Internet Browser | N/A (restricted to the system side) | - |
| Using Nintendo e-Shopping to Purchase Merchandise and Services | Purchasing Content Using ECDK | 11.2.9.6 |
| 3D Image Display | N/A (restricted to the system side) | - |
| Use of Miiverse | Posting to Miiverse and Viewing Miiverse | 11.2.9.7 |
| Sending and Receiving Photos, Images, Voice Recordings, Videos, or Text | Sending and Receiving Rich UGC | 11.2.9.2 |
| | Uploading Rich UGC (North America) | 11.2.12 |
| Internet Communication With Other Users | Data Exchange With Other Users | 11.2.9.4 |
| StreetPass Communication With Other Users | Using StreetPass | 11.2.9.5 |
| Friend Registration | Friend Registration Within the Application | 11.2.9.3 |
| Use of DS Download Play | N/A | - |
| Viewing Distributed Videos | Viewing videos obtained by communication | 11.2.9.8 |

## 11.2.9.1. Temporarily Suspending Parental Controls by Entering a PIN

When Parental Controls are in force, applications can temporarily suspend Parental Controls by having the user input a PIN. Parental Controls are then suspended if the entered PIN matches the PIN in the Parental Controls settings. Note that neither the PIN entered by the user nor the PIN in the Parental Controls settings are ever displayed on the screen.

You can compare to the PIN in the Parental Controls settings by calling the `nn::cfg::CheckParentalControlPinCode()` function.

**Code 11-17. Comparing to the Parental Controls PIN**

```
bool nn::cfg::CheckParentalControlPinCode(const char *input);
```

This function returns `true` if the string passed to *`input`* matches the PIN (four single-byte digits).

The software keyboard applet includes a mode for temporarily suspending Parental Controls. When started in this mode, the software keyboard applet operates according to a specialized sequence. The application can then determine whether to suspend Parental Controls based on

the return value from the applet.

### 11.2.9.2. Restrictions on the Transmission of Data That May Include Personal Information

Call the `nn::cfg::IsRestrictPhotoExchange()` function to check whether photo exchanges (Sending and Receiving Photos, Images, Voice Recordings, Videos, or Text) are restricted. If restricted, and so long as Parental Controls are not temporarily suspended, the application cannot exchange photos or other images with any other system. This restriction includes not only the sending and receiving of photos, but also screen images, voice recordings, videos, and text; or any data that could include private information.

**Code 11-18. Confirmation of Restrictions on the Transmission of Data That May Include Personal Information**

```
bool nn::cfg::IsRestrictPhotoExchange(void);
```

Be careful, as `true` is returned when the restriction is active.

### 11.2.9.3. Restrictions on Adding Friends

Call the `nn::cfg::IsRestrictAddFriend()` function to check whether adding friends is restricted by the Parental Controls settings. If restricted, friend registration from within an application will end in an error. In principle, the application is not permitted to temporarily cancel this restriction.

**Code 11-19. Checking Whether Adding Friends Is Restricted**

```
bool nn::cfg::IsRestrictAddFriend(void);
```

Be careful, as `true` is returned when the restriction is active.

### 11.2.9.4. Restrictions on Internet Communication With Other Users

The `nn::cfg::IsRestrictP2pInternet()` function can be used to confirm whether exchanging data or online play with other players through Internet communication has been restricted by Parental Controls. If restriction on Internet communication has been enabled, Internet communication must not be performed for the purpose of downloading user-created content or engaging in online play with other players unless the restriction is temporarily lifted.

**Code 11-20. Confirmation of Restrictions of Internet Communication With Other Users**

```
bool nn::cfg::IsRestrictP2pInternet(void);
```

Be careful, as `true` is returned when the restriction is active.

## 11.2.9.5. Restrictions on Communication via StreetPass

The `nn::cfg::IsRestrictP2pCec()` function can be used to confirm whether exchanging data with other players through StreetPass has been restricted by Parental Controls. If restriction on communication through StreetPass has been enabled, StreetPass does not work at all, and an error will occur if an attempt is made to register StreetPass data. The application cannot temporarily cancel this restriction.

**Code 11-21. Checking Restrictions on Communication via StreetPass**

```
bool nn::cfg::IsRestrictP2pCec(void);
```

Be careful, as `true` is returned when the restriction is active.

## 11.2.9.6. Restrictions on Nintendo 3DS Shopping Services

You can use the `nn::cfg::IsRestrictShopUse()` function to determine whether Nintendo 3DS Shopping Services have been restricted by Parental Controls. When they have been restricted, applications cannot add to their balance, purchase content, or take other similar actions unless the restriction is temporarily revoked.

**Code 11-22. Checking Restrictions on Nintendo 3DS Shopping Services**

```
bool nn::cfg::IsRestrictShopUse(void);
```

Be careful, as `true` is returned when the restriction is active.

## 11.2.9.7. Miiverse Restrictions

Nintendo provides a function to check whether viewing or posting to Miiverse is restricted in Parental Controls. Call `nn::cfg::IsRestrictMiiverseBrowse` to check whether viewing Miiverse is restricted. Call `nn::cfg::IsRestrictMiiversePost` to check whether posting to Miiverse is restricted.

The device Parental Controls configuration values correspond to the function's return values are described below.

**Table11-2. Relationship Between Miiverse Restrictions Configuration and Return Values**

| Restriction Name in Device Configuration | IsRestrictMiiverseBrowse | IsRestrictMiiversePost |
|---|---|---|
| "Restrict Posting and Browsing" | `true` | `true` |
| "Posting only" | `false` | `true` |
| "Do Not Restrict" | `false` | `false` |

The application cannot get Miiverse posts from other users, or post to Miiverse unless these restrictions are temporarily lifted.

### 11.2.9.8. Restriction of Video Content Acquired Through Communication

Call `nn::cfg::IsRestrictWatchVideo` to check whether Parental Controls is restricting viewing of video content acquired through communication. An application may not play back video acquired through communication when this restriction is active, unless the restriction is temporarily lifted.

**Code 11-23. Check for Restriction of Video Content Acquired Through Communication**

```
bool nn::cfg::IsRestrictWatchVideo(void);
```

Be careful, as `true` is returned when the restriction is active.

## 11.2.10. Checking for EULA Acceptance

Users must first accept the terms of the End-User Licensing Agreement (EULA) before the application can use the Nintendo 3DS Network Service, which includes Nintendo Network and StreetPass. Call the `nn::cfg::IsAgreedEula()` function to check whether the user has accepted the EULA terms. If they have not, the application must not use these networking features.

**Code 11-24. Checking for EULA Acceptance**

```
bool nn::cfg::IsAgreedEula(void);
```

The function returns `true` if the user has accepted the terms of the EULA. The FS library must be initialized before calling this function.

For more information about which features require EULA acceptance, see the Guidelines.

## 11.2.11. System-Specific ID

You can get the system-specific ID used to identify the system.

**Code 11-25. Getting the System-Specific ID**

```
bit64 nn::cfg::GetTransferableId(bit32 uniqueId);
```

For the *`uniqueId`* parameter, specify the unique 20-bit ID assigned to the application.

The system-specific ID is a 64-bit value that is guaranteed to be unique to a certain degree. This ID is also transferable in cases such as when a new replacement system has been bought. However, this ID cannot be recovered if the system was lost, stolen, or broken. The system–specific ID is changed when the user runs format system memory, and the original ID cannot thereafter be recovered.

Possible uses of this system ID include the following.

- Ensuring that initial values of application parameters are different for each system.
- Creating data accessible only from one specific system.
- The identifier or its seed for use in local communication.

**Note:** When save data is created, the system-specific ID is saved, and subsequently the use of that saved system-specific ID in the save data can support changes such as those resulting from repairs. In addition, upon saving, a 128-bit value that matches the current time is saved, which can prevent the same data from being saved to multiple cards.

If you are using the ID to create data accessible only from that system, note that this data will be inaccessible if the system is formatted with format system memory, lost, or stolen.

## 11.2.12. Restrictions Based on COPPACS

The COPPA Compliance System (COPPACS) is a system provided by Nintendo to enable applications created for North America (if the system region is North America, and the country setting is U.S. or Canada) to comply with the Children's Online Privacy Protection Act (COPPA).

**Note:** For a list of applications subject to COPPACS, see the UGC section of the guidelines. Plans are to provide details about COPPACS in the future in the System Application and Applet Specifications.

You can determine whether restrictions based on COPPACS are in effect.

**Code 11-26. Getting Restrictions Based on COPPACS**

```
bool nn::cfg::IsCoppacsSupported();
nn::cfg::CfgCoppacsRestriction nn::cfg::GetCoppacsRestriction(void);
```

You can confirm the support method for COPPACS by calling the `nn::cfg::GetCoppacsRestriction()` function.

When `CFG_COPPACS_RESTRICTION_NONE` is returned, no restrictions are in effect, so COPPACS support is unnecessary.

When `CFG_COPPACS_RESTRICTION_NEED_PARENTAL_PIN_CODE` or `CFG_COPPACS_RESTRICTION_NEED_PARENTAL_AUTHENTICATION` is returned, restrictions are in effect, so COPPACS support must be provided.

The former can be released by entering the parental control authentication number. Enter and check the authentication number within the application. If the correct authentication number is entered, a temporary suspension can be performed (see 11.2.9.1. Temporarily Suspending Parental Controls by Entering a PIN). The latter cannot be released from within the application. The COPPACS authentication procedure within the system settings must be used.

Note that you must temporarily exit the application when performing the authentication procedure for this setting. You can confirm the setting by restarting the application when returning to the

application from System Settings. When continuing a process which was being performed before jumping to System Settings, always reconfirm the support method for COPPACS. In some cases, when reconfirming, you may be asked to perform the authentication procedure within System Settings.

To simply confirm which countries are subject to COPPACS with the current System Settings, you can use the `nn::cfg::IsCoppacsSupported()` function. Note that if the system setting is for one of the countries on the list, the value `true` is returned even if COPPACS restrictions have not been enabled.

> **Note:** A description of how to jump to the COPPACS authentication procedure screen (`PARENTAL_CONTROLS_COPPACS`) is provided in 5.3.7. Jump to System Settings. For implementation examples, see the sample demo (`coppacs` in `cfg`).

## 11.3. Finalizing

Call the `nn::cfg::Finalize()` function when done using the CFG library.

**Code 11-27. Finalizing the CFG Library**

```
void nn::cfg::Finalize(void);
```

If this function is called before initialization, nothing will happen.

The number of times the library's initialization function has been called is recorded. Until the finalize function is called the same number of times, the library remains in use.



# 12. Applets

This chapter describes the libraries that are required in order to use the applets provided by the 3DS system.

The library provided by each applet can be used to start the applet from an application and use its features. The system generally allocates the memory required to start applets, but some applets require that the application pass the working memory. In addition, the application effectively stops while applets are running because they use the same CPU as the application after the operations of the called thread are stopped.

## 12.1. Library Applets

Features often used by applications are provided with 3DS as the following library applets.

- Software Keyboard Applet
- Photo Selection Applet
- Mii Selection Applet
- Sound Selection Applet
- Error/EULA Applet
- Circle Pad Pro Calibration Applet
- EC Applet
- Login Applet

**Note:** The Mii Selection applet is provided in the CTR Face Library package.

## 12.1.1. Information Common to All Library Applets

The processing required to start and return from a library applet is basically the same as that used to start and return from the HOME Menu. Likewise, while a library applet is running, there are similar restrictions on using devices such as getting key input and rendering, and only the thread that called the library applet will stop it. Try not to execute unnecessary threads while library applets are executing. When creating threads that will continue to operate even while a library applet is running, note that the library applet creates threads with the priority settings shown in the following table. Set your thread priority so that it does not affect these thread processes.

**Table 12-1. Priorities of Threads Created by Library Applets**

| Library Applet | Priority | Comments |
|---|---|---|
| (Common to all) | 15 | Used for notifications such as sleep notifications. |
| Software Keyboard Applet | 17 to 20 | |
| Photo Selection Applet | 16, 18, 20, 21, 25 | |
| Mii Selection Applet | 17 | |
| Sound Selection Applet | 16, 18 | |
| Error/EULA Applet | 17, 20 | |
| Circle Pad Pro Calibration Applet | 17 to 19 | |
| EC Applet | 17, 22 | Used for communication and related functions. |
| Login Applet | 17, 22 | Used for communication and related functions. |

The application must support close processing when `nn::applet::IsExpectedToCloseApplication()` returns `true` after control returns to the application in the same way as it returns to the application when `nn::applet::WaitForStarting()` returns `true` while calling the function after the HOME Menu starts. However, when the POWER Button is pressed while the library applet is being displayed, the application transitions to a state where it has no rendering rights and at the same time where `nn::applet::IsExpectedToProcessPowerButton` returns `true`. If the application responds to the close request in this case, the screen refresh will halt during close processing. Please handle

the POWER Button first in your implementation.

#### 12.1.1.1. Recovering From Library Applets

When the HOME Button, the POWER Button, or the software reset button combination (L + R + START) is pressed while a library applet is starting, the application immediately recovers and the following behavior results.

**Table 12-2. Behavior When Recovering From Library Applets**

| Button | Behavior |
|---|---|
| HOME Button | Returns a value as the return code indicating that the HOME Button was pressed. After recovery, `nn::applet::IsExpectedToProcessHomeButton` returns `true`. |
| POWER Button | Returns a value as the return code indicating that the POWER Button was pressed. After recovery, the `nn::applet::IsExpectedToProcessPowerButton()` function returns `true`. |
| Software Reset | Returns a value as the return code indicating that there was a software reset. |

#### 12.1.1.2. Preloading

Library applets that support preloading can perform processes such as loading in advance. As a result, you can shorten the time that the screen freezes between the function call that starts the library applet and its display on the screen.

The names of the functions for preloading differ for each library applet, but they follow the same basic standards. Functions that begin the preloading process start with `Preload`. Functions that wait for preloading to complete start with `WaitForPreload`. And functions that cancel the preloading process start with `CancelPreload`.

When starting a preloaded library applet, you must wait for preloading to complete before starting. Note also that if you call a function that waits for completing without having first called a function that begins preloading, control never returns. However, this caution applies when both function calls take place on one thread. If the waiting-for-completion thread is different from the thread that begins preloading, control returns even if the order of the calls is reversed. Displaying the HOME Menu using `nn::applet::ProcessHomeButton` cancels preloading. If you navigate to the HOME Menu between calls to `Preload` and `WaitForPreload`, control does not return from `WaitForPreload`.

You cannot preload multiple library applets at the same time. To start a library applet other than one you have preloaded, you must first cancel the preload. Otherwise, after a library applet is started the preload is released, so there is no need to cancel the preload after recovery from a library applet.

### 12.1.2. Software Keyboard Applet

This library applet is called when an application requires text input from the user, such as for a user name or password. It displays a software keyboard on the lower screen. It includes several features

such as entering passwords, restricting the number and type of characters that can be entered, and filtering prohibited words. On the upper screen, you can darken or continue displaying the application screen that is displayed when the software keyboard was started.

To use this library applet, you must include the `nn/swkbd.h` header file and add the `libnn_swkbd` library file.

Parameters passed when starting the library applet are defined by the `nn::swkbd::Parameter` data structure. Detailed operational settings are made with the config (`nn::swkbd::Config`) member of parameters. However, initialization is required before configuration.

**Code 12-1. Initializing Software Keyboard Operational Settings**

```
void nn::swkbd::InitializeConfig(nn::swkbd::Config* pConfig);
```

The `Config` structure passed in *`pConfig`* is initialized to the default settings. For operational settings that can be made in members of the `Config` structure, see the *Applet Specifications*.

The application must allocate work memory for this library applet. The size of the work memory required differs depending on the operational settings, and can be obtained using the `nn::swkbd::GetSharedMemorySize()` function.

**Code 12-2. Getting the Work Memory Size for the Software Keyboard**

```
s32 nn::swkbd::GetSharedMemorySize(
                const nn::swkbd::Config* pConfig,
                const void* pInitialStatusData = DELETEME,
                const void* pInitialLearningData = DELETEME);
```

Pass a pointer to the `Config` structure used to make operational settings in *`pConfig`*.

If the operational settings of the software keyboard being used the last time it was run have been saved, you can restore that previous state by passing the start address for that data in *`pInitialStatusData`*. If this data does not need to be restored or it was not saved, specify `NULL`.

If training data for predictive text input has been saved with the operational settings of the software keyboard that were used the last time it was run, the training state last in effect can be restored by passing the start address for that data in *`pInitialLearningData`*. If this data does not need to be restored or it was not saved, specify `NULL`.

The start address of work memory must be allocated with `nn::swkbd::MEMORY_ALIGNMENT` (4096-byte) alignment and its size must be a multiple of `nn::swkbd::MEMORY_UNITSIZE` (4096 bytes). **Memory allocated from device memory must not be specified for the work memory.**

After parameters have been set and work memory allocated, you can start the software keyboard using the `nn::swkbd::StartKeyboardApplet()` function.

**Code 12-3. Starting the Software Keyboard Applet**

```
bool nn::swkbd::StartKeyboardApplet(
                nn::applet::AppletWakeupState* pWakeupState,
                nn::swkbd::Parameter* pParameter,
                void* pSharedMemoryAddr,
                size_t sharedMemorySize,
                const wchar_t* pInitialInputText = DELETEME,
                const nn::swkbd::UserWord* pUserWordArray = DELETEME,
                const void* pInitialStatusData = DELETEME,
                const void* pInitialLearningData = DELETEME
                nn::applet::AppTextCheckCallback callback = DELETEME);
```

In *pParameter*, specify a pointer to the Parameter structure, which has the Config structure used to configure operational settings as one of its members. Information such as the input text string is stored in the structure specified by this argument.

Specify the start address and size of work memory in *pSharedMemoryAddr* and *PSharedMemorySize*.

If a UTF-16LE text string that is not NULL is passed in *pInitialInputText*, the software keyboard is run with the specified text string set in the input field.

In *pUserWordArray*, specify an array of words to register in the user dictionary.

Specify the same value obtained for the work memory size in *pInitialStatusData* and *pInitialLearningData*.

In *callback*, specify the callback function to use when the application is checking an input string. When the application does not perform a check due to its operation settings, this argument is ignored.

Startup fails if the return value of this function is false. Startup succeeds if the return value is true.

The software keyboard thread operates at a priority between 17 and 20. So even while the software keyboard is running, attention must be paid to the priority settings of threads that continue to operate.

## 12.1.3. Photo Selection Applet

This library applet is used to select the data for one photo from among those registered in the Nintendo 3DS camera album for use by the application. You can specify extraction conditions such as the capture time and type when starting the applet. You can also filter the data to be listed. Note that the applet can only select photos that are saved on an SD card.

To use this library applet, you must include the nn/phtsel.h header file and add the libnn_phtsel library file.

Parameters to be passed when this library applet is started are defined in the nn::phtsel::Parameter structure. Basic operational settings are made in the m_config member (nn::phtsel::Config structure). Detailed settings such as extraction conditions are made in the m_input member (nn::phtsel::PhtselInput structure). The execution result is stored in the m_output member (nn::phtself::PhtselOutput structure). For the available settings and other information, see the sample demo.

The application must allocate work memory when the applet background displays images captured by the application. The size required by work memory can be obtained using the nn::phtsel::GetWorkBufferSize() function.

**Code 12-4. Getting the Work Memory Size for Photo Selections**

```
size_t nn::phtsel::GetWorkBufferSize();
```

The start address of work memory must be allocated with 4096-byte alignment and its size must be a multiple of 4096 bytes. **Memory allocated from device memory must not be specified for the work memory.**

After the parameters have been set and the work memory allocated, you can start the photo selection library applet using the nn::phtsel::StartPhtsel() function. Call the nn::phtsel::StartPhtselNoCapture() function if the applet background does not display

images captured by the application.

**Code 12-5. Starting the Photo Selection Library Applet**

```
nn::applet::AppletWakeupState nn::phtsel::StartPhtsel(
                nn::phtsel::Parameter* pParameter, void* pWorkBuffer);
nn::applet::AppletWakeupState nn::phtsel::StartPhtselNoCapture(
                nn::phtsel::Parameter* pParameter);
```

In *pParameter*, specify the pointer to the Parameter structure for which settings were performed. Specify the start address of work memory in *pWorkBuffer*.

Execution results are stored in the m_output member of *pParameter* upon return to the application.

## 12.1.4. Mii Selection Applet

This library applet is called when using Mii characters in an application to select from among registered Mii characters. You can select only one registered Mii or one guest Mii. (Six are available by default.) You can also select whether to display a list of guest Mii characters and whether to display the operating screen in the upper or lower screen.

**Note:** To use the Mii Selection applet from inside your application, you must use the CTR Face Library (middleware).

For instructions on using this library, see the CTR Face Library documentation.

## 12.1.5. Sound Selection Applet

This library applet is called when selecting sound data recorded using Nintendo 3DS Sound for a sound to be used, such as an application sound effect. Only one instance of sound data can be selected. This applet cannot be used to record sounds, sort data, delete data, or otherwise manipulate data. Note that the applet can only select sound data that is saved on an SD card.

To use this library applet, you must include the header file (nn/voicesel.h) and add the library file (libnn_voicesel).

The nn::voicesel::Parameter structure defines the parameters to pass into the library applet when starting it. Use the parameter's config member (nn::voicesel::Config structure) to configure basic operations, and the input member (nn::voicesel::Input structure) to configure advanced features, such as extraction conditions. The results are stored in the output member (nn::voicesel::Output structure). For the available settings and other information, see the sample demo.

After you are finished configuring the parameters, you can start the sound selection library applet by calling nn::voicesel::StartVoiceSel.

**Code 12-6. Starting the Sound Selection Applet**

```
nn::applet::AppletWakeupState nn::voicesel::StartVoiceSel(
                nn::voicesel::Parameter* pParameter);
```

In *pParameter*, specify the pointer to the Parameter structure for which settings were performed.

When control returns to the application, *pParameter* 's output member will store the results of execution.

## 12.1.6. Error/EULA Applet

When using wireless-based communication features such as StreetPass, you must check the EULA (the licensing agreement related to Nintendo 3DS network services) and gain the consent of the user. If an application that uses these communication features has not confirmed that the user has agreed to the latest EULA and gained the user's consent, you can use this library applet to display the EULA and get consent. Acceptance of the EULA is required in order to use communication features, but applications are not required to display the EULA. If the application does not display the EULA, display an error at the time of communication and guide the user to System Settings.

Not only does this library applet display the EULA, it can also display error messages. As long as infrastructure communication-related libraries (such as AC and FRIENDS) are being used, the correct corresponding error message is displayed when an error code is passed to the applet. You can also display proprietary error messages from the application.

To use this library applet, you must include the nn/erreula.h header file and add the libnn_erreula library file.

Parameters to be passed when this library applet is started are defined in the nn::erreula::Parameter structure. Error codes, error messages, and operational settings are made in the config member (nn::erreula::Config structure). Be sure to configure the setting after initializing with the nn::erreula::InitializeConfig() function. For more information about parameter settings, see the *Applet Specifications* and sample demos.

After you set the parameters and allocate the work memory, you can start the Error/EULA applet using the nn::erreula::StartErrEulaApplet() function.

**Code 12-7. Starting the Error/EULA Applet**

```
void nn::erreula::StartErrEulaApplet(
                nn::applet::AppletWakeupState* pWakeupState,
                nn::erreula::Parameter* pParameter);
```

In *pParameter*, specify the pointer to the Parameter structure for which settings were performed.

If the EULA display is made by the setting, the agreement sequence of the Network Services Agreement is displayed and the result of agreement or disagreement to the Network Services Agreement is returned to the application.

All of the infrastructure communication features associated with the application, and StreetPass communication, return the EULA disagreement error if the user does not agree to the Network Services Agreement in the version that the application requires. NEX login, registration of download tasks, creation of StreetPass box, etc., will all be subject to the agreement. The version of the Network Services Agreement that requires an agreement will automatically be embedded in ROM. Be sure to confirm whether the Network Services Agreement is required with the nn::cfg::IsAgreedEula() function, and call back this applet if the agreement is not accepted. Only when the applet is called back in the state of not agreeing to the Network Services Agreement, will "EULA agreement" or "EULA disagreement" be returned correctly.

## 12.1.7. Circle Pad Pro Calibration Applet

This library applet is used for calibrating the feel of controls on the Right Circle Pad installed in the Circle Pad Pro. Applications supporting the Circle Pad Pro must provide a start scene for this applet so that users can calibrate the feel of controls on the Right Circle Pad.

> **Note:** The C Stick on SNAKE functions like the Right Circle Pad on a permanently connected Circle Pad Pro. If an application running on SNAKE calls the Circle Pad Pro Calibration Applet, a message describing how to calibrate the C Stick is displayed.

To use this library applet, you must include the header file (`nn/extrapad.h`). In addition, if you are using CTR-SDK 11.3.x or earlier, you must also add the library file (`libnn_extrapad`).

The parameter passed when the library applet is started is defined by the `nn::extrapad::Parameter` structure. Operating settings are performed in the *`config`* parameter member variables (`nn::extrapad::Config` structure). Always perform initialization with the `nn::extrapad::InitializeConfig()` function before performing settings for whether to support the HOME Button and software reset.

When parameter setting procedures are complete, you can start the library applet with the `nn::extrapad::StartExtraPadApplet()` function.

**Code 12-8. Starting the Circle Pad Pro Calibration Applet**

```
void nn::extrapad::StartExtraPadApplet(
                nn::applet::AppletWakeupState* pWakeupState,
                nn::extrapad::Parameter* pParameter);
```

In *`pParameter`*, specify the pointer to the `Parameter` structure for which settings were performed.

## 12.1.8. EC Applet

The EC applet is a library applet for purchasing and managing downloadable content and service items.

> **Note:** For more information about EC features, see the *CTR-SDK API Reference*.

## 12.1.9. Login Applet

This library applet communicates with the account server, working on behalf of the application to authenticate the account and get various service tokens and the like.

> **Note:** For more information about the login applet, see the *3DS Programming Manual: Wireless Communication*.

# 12.2. System Applets

A system applet is generally an applet that is started from the HOME Menu, but libraries are provided to start and use the applets from an application. The processing required to start and return from a system applet from an application is basically the same as that used to start and return from the HOME Menu.

## 12.2.1. Internet Browser Applet

The WEBBRS library allows you to start the built-in Internet browser by specifying a URL from an application. The startup method is similar to that of a library applet, but it differs in that closing the Internet browser that was started results in returning to the HOME Menu with the application suspended.

> **Warning:** Depending on which system updates have been applied, there may be systems that do not have the Internet browser installed. For this reason, you must check whether the Internet browser is installed on the system before starting it.

> **Note:** For more information about the WEBBRS library, see the *CTR-SDK API Reference*.

## 12.2.2. Miiverse Application and Post App

The OLV library enables the use of Miiverse features in an application. This makes it possible to start the Miiverse application or Post app and receive post data from within an application. These are referred to as "applications" but are actually included with the system applets.

> **Note:** For more information about the OLV library, see the *CTR-SDK API Reference*.

### 12.2.2.1. Starting an Application From the Miiverse Post Page

Applications can be started from a post in Miiverse.

The application must be configured as follows to enable this feature.

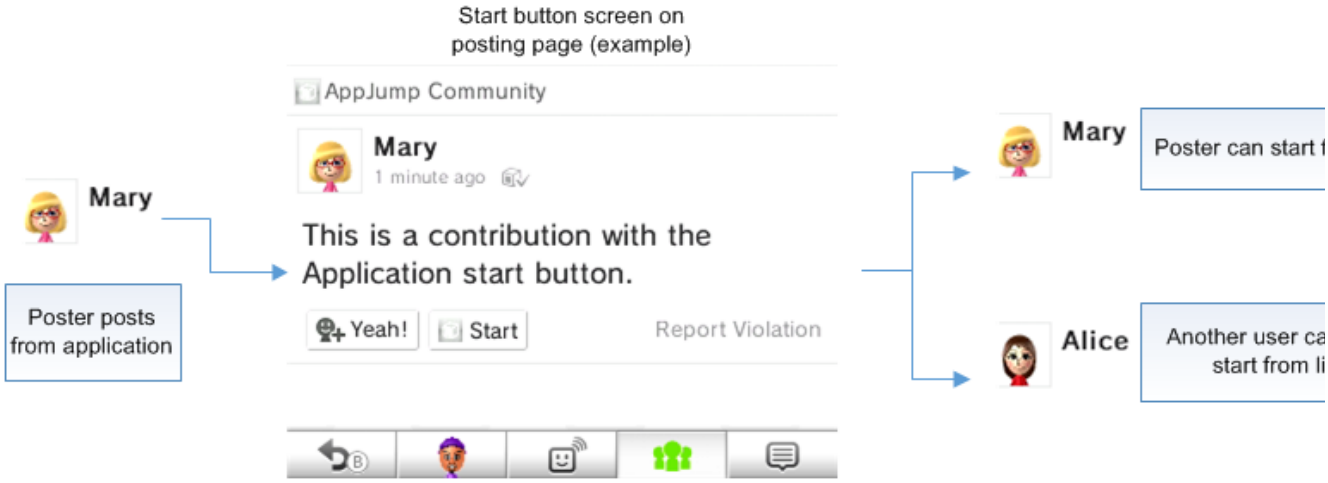
- Specify `EnableMiiverseJumpArgs:True` in the BSF file.
- Specify `FLAG_APP_STARTABLE` flag with the `nn::olv::UploadPostDataByPostApp()` function when posting.

**Operation in the Miiverse Application**

When you view a post with this flag set in the Miiverse application, a **Launch** button appears in

the post. Any user who owns the application can use this button to start it. Users who do not own the application can view the post but cannot start the application.

**Figure 12-1. Starting an Application From the Post Page**






# 13. Supplemental Libraries

This chapter describes supplemental libraries provided for using internal system resources, such as power and the pedometer, and shared resources such as internal fonts.

## 13.1. PTM Library

The PTM library is provided for getting power-related information and using RTC-based alarms.

### 13.1.1. Initializing and Finalizing

The PTM is initialized and finalized using the `nn::ptm::Initialize()` function and the `nn::ptm::Finalize()` function.

**Code 13-1. Initializing and Finalizing the PTM Library**

```
nn::Result nn::ptm::Initialize();
nn::Result nn::ptm::Finalize();
```

Neither function returns an error as long as the hardware is not damaged.

## 13.1.2. Getting Power-Related Information

The following power-related information can be obtained: connection status of the power adapter, charge status of the battery, and the battery level.

**Code 13-2. Getting Power-Related Information**

```
nn::ptm::AdapterState       nn::ptm::GetAdapterState();
nn::ptm::BatteryChargeState nn::ptm::GetBatteryChargeState();
nn::ptm::BatteryLevel       nn::ptm::GetBatteryLevel();
```

You can get the connection status of the power adapter by calling the `nn::ptm::GetAdapterState` function. The following table shows the possible return values.

**Table 13-1. nn::ptm::AdapterState Enumerator**

| Values | Description |
|---|---|
| ADAPTERSTATE_NOCONNECTED | Power adapter not connected |
| ADAPTERSTATE_CONNECTED | Power adapter connected |

You can get the battery charge status by calling the `nn::ptm::GetBatteryChargeState()` function. The following table shows the possible return values.

**Table 13-2. nn::ptm::BatteryChargeState Enumerator**

| Values | Description |
|---|---|
| BATTERYCHARGESTATE_NOCHARGING | Not charging |
| BATTERYCHARGESTATE_CHARGING | Charging |

You can get the battery level by calling the `nn::ptm::GetBatteryLevel()` function. The following table shows the possible return values.

**Table 13-3. nn::ptm::BatteryLevel Enumerator**

| Values | Description |
|---|---|
| BATTERYLEVEL_0 (BATTERYLEVEL_MIN) | Battery level is 0% |
| BATTERYLEVEL_1 | Battery level is 1% to 5% |
| BATTERYLEVEL_2 | Battery level is 6% to 10% |
| BATTERYLEVEL_3 | Battery level is 11% to 30% |
| BATTERYLEVEL_4 | Battery level is 31% to 60% |
| BATTERYLEVEL_5 (BATTERYLEVEL_MAX) | Battery level is 61% to 100% |

## 13.1.3. Alarms Using the RTC

For information about this feature, see 8.5.2. RTC Alarm Feature.

# 13.2. PL Library

The PL library is provided so that applications can use features and resources built into the 3DS system such as the pedometer and internal (shared) fonts. Although there are no functions for initializing or finalizing the PL library itself, other libraries may need to be initialized in order to use these features and resources.

## 13.2.1. Pedometer

Pedometer information is recorded in terms of the number of steps for each hour for up to 120 months (approximately 10 years). Because pedometer information is stored in the power-related module, you must initialize the PTM library ahead of time in order to access pedometer information. For information about initializing the PTM library, see 13.1.1. Initializing and Finalizing.

Use the following functions to get pedometer-related information.

**Code 13-3. Functions for Getting Pedometer Information**

```
bool nn::pl::GetPedometerState();
u32 nn::pl::GetTotalStepCount();
s8 nn::pl::GetStepHistoryEntry(nn::pl::PedometerEntry* pEntry);
void nn::pl::GetStepHistory(u16 pStepCounts[], s32 numHours,
                            nn::fnd::DateTime start);
nn::Result nn::pl::GetStepHistoryAll(nn::pl::PedometerHistoryHeader& header,
                                     nn::pl::PedometerHistoryData& data);
```

The `nn::pl::GetPedometerState()` function returns whether the accelerometer is functioning as a pedometer. The accelerometer functions as a pedometer when the power is on and the system is closed. Applications therefore do not need to track whether the accelerometer is functioning as a pedometer because they have usually transitioned into Sleep Mode.

The `nn::pl::GetTotalStepCount()` function returns the number of steps accumulated so far.

The `nn::pl::GetStepHistoryEntry()` function stores the year and month of recorded step count entries in the `nn::pl::PedometerEntry` array specified by *`pEntry`* and returns the number of entries stored as its return value. Allocate enough memory for the array specified by *`pEntry`* so that the maximum number of entries (`NUM_MONTHHISTORIES`) can be stored.

The `nn::pl::GetSetpHistory()` function stores step count information for each hour for the number of hours specified by *`numHours`* (future direction only) starting from and including the hour specified by *`start`* in the array specified by *`pStepCounts`*. Zero is stored for hours for which there is no step count information.

The `nn::pl::GetHistoryAll()` function is used to get all step count information. The step count information obtained is split into header information and data. The order of data corresponds to the order of header information, but header information is not necessarily arranged in order of year and month. The year and month associated with step count information can be confirmed using `monthInfo` included in header information. If `INVALID_COUNTER` is stored in `unusedCounter` for an entry, it indicates that the data is invalid and no step count data has been recorded for that entry.

**Note:** CTR-SDK includes the PedometerChanger development tool, which lets you view and manipulate the pedometer information.

## 13.2.2. Internal Fonts

Internal fonts are provided as shared resources that can be accessed by applications. The type of internal fonts loaded as standard differs depending on the region set for the system.

The types of internal fonts that can be used are defined as shown below by the `nn::pl::SharedFontType` enumerator type.

**Table 13-4. nn::pl::SharedFontType Enumerator**

| Values | Description |
|---|---|
| `SHARED_FONT_TYPE_STD` | European fonts. Standard for the Japanese, European, and United States regions. |
| `SHARED_FONT_TYPE_CN` | Chinese fonts. Standard for the Chinese region. |
| `SHARED_FONT_TYPE_KR` | Korean fonts. Standard for the Korean region. |
| `SHARED_FONT_TYPE_TW` | Taiwanese fonts. Standard for the Taiwanese region. |

### 13.2.2.1. Loading Internal Fonts

Internal fonts are loaded using the `nn::pl::InitializeSharedFont()` function. Although fonts for all regions are built into systems for any region, the `InitializeSharedFont()` function only loads those fonts associated with the system region.

**Code 13-4. Loading Internal Fonts**

```
nn::Result nn::pl::InitializeSharedFont();
nn::pl::SharedFontLoadState nn::pl::GetSharedFontLoadState();
```

Calling the `nn::pl::InitializeSharedFont()` function starts loading internal fonts. However, loading of internal fonts may not yet be complete when the function finishes executing. The return value (`nn::pl::SharedFontLoadState`) of the `nn::pl::GetSharedFontLoadState()` function can be used to verify that fonts have finished loading.

**Table 13-5. nn::pl::SharedFontLoadState Enumerator**

| Values | Description |
|---|---|
| `SHARED_FONT_LOAD_STATE_NULL` | Load not started. |
| `SHARED_FONT_LOAD_STATE_LOADING` | Loading. |
| `SHARED_FONT_LOAD_STATE_LOADED` | Load complete. |
| `SHARED_FONT_LOAD_STATE_FAILED` | Load failed. |

You can get the start address, size, and font type of loaded font data using the `nn::pl::GetSharedFontAddress`, `nn::pl::GetSharedFontSize`, and `nn::pl::GetSharedFontType()` functions, respectively.

**Code 13-5. Getting Loaded Font Data Information**

```
void*                   nn::pl::GetSharedFontAddress();
size_t                  nn::pl::GetSharedFontSize();
nn::pl::SharedFontType  nn::pl::GetSharedFontType();
```

If you want to display characters only included in the internal font of a region other than the system region to, for example, display messages sent from a system of another region, you must mount the internal font archive using the `nn::pl::MountSharedFont()` function and have the application load the corresponding internal font.

**Code 13-6. Loading Internal Fonts (Other Regions)**

```
nn::Result nn::pl::MountSharedFont(const char* archiveName,
        nn::pl::SharedFontType sharedFontType, size_t maxFile,
        size_t maxDirectory, void* workingMemory, size_t workingMemorySize);
nn::Result nn::pl::UnmountSharedFont(const char* archiveName);
nn::Result nn::pl::GetSharedFontRequiredMemorySize(
        s32* pOut, nn::pl::SharedFontType sharedFontType, size_t maxFile,
        size_t maxDirectory);
```

*`archiveName`* specifies the archive name of the font archive to be mounted.

*`sharedFontType`* specifies the type of internal font to be mounted. The files loaded differ depending on the type of internal font specified here. If the specified font does not exist, `nn::pl::ResultSharedFontNotFound` is returned as the return value.

**Table 13-6. Internal Font Types and Associated Filenames (When "font" Is Specified in the Archive Name)**

| Internal Font Type | Filename |
|---|---|
| SHARED_FONT_TYPE_STD | font:/cbf_std.bcfnt.lz |
| SHARED_FONT_TYPE_CN | font:/cbf_zh-Hans-CN.bcfnt.lz |
| SHARED_FONT_TYPE_KR | font:/cbf_ko-Hang-KR.bcfnt.lz |
| SHARED_FONT_TYPE_TW | font:/cbf_zh-Hant-TW.bcfnt.lz |

Font files are compressed in LZ77 format and should be decompressed using the CX library before using them as fonts.

Specify the number of files and directories that can be opened simultaneously in *`maxFile`* and *`maxDirectory`*, respectively. Pass the working memory and its size to *`workingMemory`* and *`workingMemorySize`*. Use the `nn::pl::GetSharedFontRequiredMemorySize()` function to get the size of the working memory required.

Unmount the archive using the `nn::pl::UnmountSharedFont()` function after font files have finished loading.

## 13.2.2.2. Using Font Data

The same format as BCFNT files converted using `ctr_FontConverter` is used for internal font data. The `nn::font::ResFont` class of the FONT library must be used to display internal fonts.

For information about how to display fonts using the FONT library, see the API Reference and sample demos.

## 13.2.3. Play Coins

Play Coins accumulate as the user walks around with their system. You are free to make use of these accumulated Play Coins. For example, users might exchange Play Coins for bonus items in your application.

To use Play Coins, you must include the header file `nn/pl/CTR/pl_GameCoin.h` and add the library file `libnn_plCoin`.

> **Note:** CTR-SDK includes a development tool (PlayCoinSetter) for setting the number of Play Coins a system has.

### 13.2.3.1. Initialization and Finalization

The initialization and finalization of the library are performed by calling the `nn::pl::InitializeGameCoin()` and `nn::pl::FinalizeGameCoin()` functions.

**Code 13-7. Initializing and Finalizing the Play Coin Library**

```
void nn::pl::InitializeGameCoin();
void nn::pl::FinalizeGameCoin();
```

Before initializing the Play Coin library, you must initialize the FS and PTM libraries. Because there is no support for nested initialization calls, even if you call the initialization function multiple times, calling the `nn::pl::FinalizeGameCoin()` function once will finalize use of the library.

### 13.2.3.2. Getting the Number of Play Coins Held

You can get the number of Play Coins held by the system with the `nn::pl::GetGameCoinCount()` function.

**Code 13-8. Getting the Number of Play Coins Held**

```
nn::Result nn::pl::GetGameCoinCount(u16* pCount);
```

The currently held number of Play Coins is stored in *`pCount`*.

This function has a high load and entails a write to system NAND memory. Other than when starting the application, call it only when there is a possibility that the number of Play Coins might have increased or decreased, such as when recovering from Sleep Mode, the HOME Menu, or a library applet.

When this function returns `nn::pl::ResultGameCoinDataReset`, it indicates that the Play Coins have been reset due to corrupt data or some other reason, but it is handled as a success (meaning that the `IsSuccess()` function returns `true`). When informing the user that the Play Coins have been reset, check whether this value has been returned.

### 13.2.3.3. Spending Play Coins

Play Coins are spent with the `nn::pl::UseGameCoin()` function.

**Code 13-9. Spending Play Coins**

```
nn::Result nn::pl::UseGameCoin(u16* pCount, u16 useCount);
```

When the process is successful, the number of Play Coins after spending is stored in *`pCount`*. When the process fails, an undefined value is stored there.

Specify the number of Play Coins to spend in *`useCount`*. When the number specified is more than the number held, no Play Coins are spent, and `nn::pl::ResultLackOfGameCoin` is returned to indicate that there were not enough Play Coins.

This function's load is high and it entails a write to system NAND memory.



# 14. Infrared Communication Between Systems

This chapter describes communication between systems using the IR library, which allows applications to use the CTR infrared communications module.

**Note:** CTR-SDK includes a development tool (IrCommunicationChecker) for checking the information communicated between systems by infrared communication.

**Warning:** If you are developing an extended SNAKE application that uses infrared for communication between systems, make sure that you test communication between CTR and SNAKE to verify that the processing speed differences do not cause any communication problems.

**Warning:** If you want to use infrared communication from an application while another feature is using infrared communication, you must end the other feature first.

Infrared communication is used by the following features.

- Circle Pad Pro
- NFP (Only CTR)

## 14.1. nn::ir::Communicator Class

Functions for communication between systems using the infrared communications module are all defined as static functions of the `nn::ir::Communicator` class.

> **Warning:** The `nn::ir::Communicator` class only supports 1:1 communication between systems. It cannot be used for communicating with devices other than 3DS systems, or for communicating among multiple CTR systems.

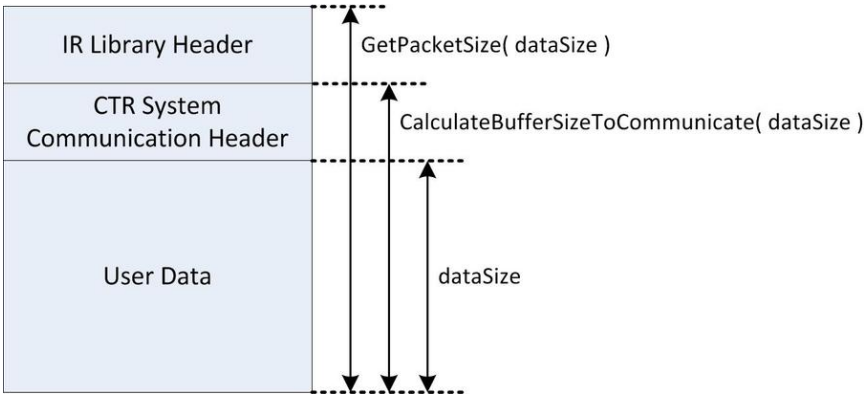
## 14.1.1. Security

The library handles encryption and guarantees completeness of transmitted data. Each sent packet also includes a sequence number that is incremented with each send to prevent resend attacks.

## 14.1.2. Packets

Packets transmitted by the IR library include an IR library header, which is used for communication control, and a system communication header, which includes security and other information.

**Figure 14-1. IR Library Packet**




Get the size of the user data in a packet after appending header information using the following functions.

**Code 14-1. Functions to Get User Data Size From Data With Header Information Appended**

```
size_t nn::ir::Communicator::GetPacketSize(size_t dataSize);
size_t nn::ir::Communicator::CalculateBufferSizeToCommunicate(size_t dataSize);
```

The user data size is specified in *`dataSize`*.

The `GetPacketSize()` function calculates the size required to save the packet, and the `CalculateBufferSizeToCommunicate()` function calculates the space required for send and receive buffers.

## 14.1.3. Support for Sleep Mode

When transitioning to Sleep Mode, connections are interrupted and any processing is forcibly stopped. Unsent packets and unreceived data are all discarded.

When a connection is interrupted due to Sleep Mode, no request to disconnect is sent to the partner. When sending a disconnect notification to the partner, perform this processing before transitioning to Sleep Mode, and only transition to Sleep Mode when the disconnection is confirmed.

### 14.1.4. Communication Range

The following table shows the infrared communication range for two 3DS systems placed with the infrared receivers (on the back of the units) directly facing each other.

**Table 14-1. Communication Range for Infrared Communication**

| Item | Range |
|---|---|
| Distance | 0 to 20 cm from the infrared receiver when the lower screen (Touch Screen) is horizontal |

## 14.2. Initialization

The IR library is initialized by calling the `Initialize()` function of the `nn::ir::Communicator` class.

**Code 14-2. IR Library Initialization Function**

```
nn::Result nn::ir::Communicator::Initialize(
        void* pBuf, size_t bufSize,
        size_t receiveBufferDataSize, size_t receiveBufferManagementSize,
        size_t sendBufferDataSize, size_t sendBufferManagementSize);
```

Specify the buffer used by the library to manage transmitted packets, the buffer size in *pBuf*, and *bufSize*. The buffer passed to the library must be allocated by the application beforehand. The starting address must be `nn::ir::Communicator::BUFFER_ALIGNMENT` (4096-byte) aligned, and its size must be a multiple of `nn::ir::Communicator::BUFFER_UNITSIZE` (4096-byte). **Buffers allocated from device memory cannot be used.**

Buffers allocated from device memory cannot be used. The buffer passed to the library is divided into several areas, and the sizes of each area are specified in *receiveBufferDataSize*, *receiveBufferManagementSize*, *sendBufferDataSize*, and *sendBufferManagementSize*. For more information about specifying sizes, see 14.2.1. Buffer Passed to the Library.

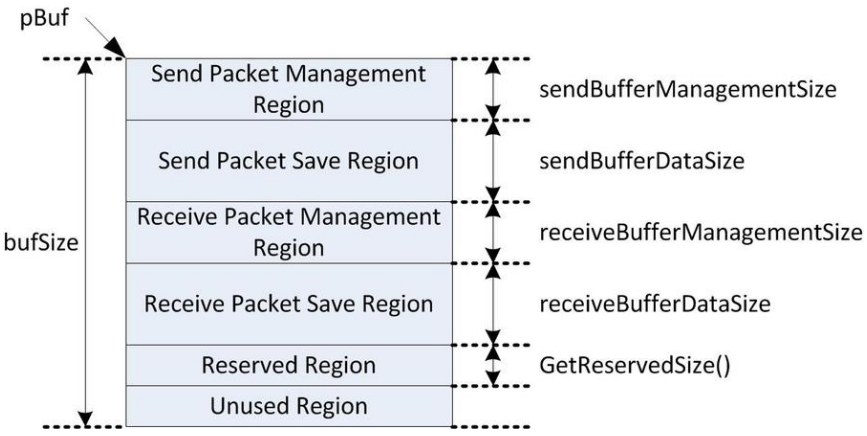
The communication speed is fixed at 115200 bits per second. In 3DS infrared communication, a start bit and an end bit are added to each byte sent. Consequently, the transmission rate for raw data is the baud rate value multiplied by 0.8, which is 92160 bits per second (11520 bytes/s).

### 14.2.1. Buffer Passed to the Library

IR library transmission processes are implemented with asynchronous functions. The buffer passed to the library during initialization is used mainly for temporary storage of sent packets.

The buffer passed to the library is divided into the regions shown in the following figure.

**Figure 14-2. Buffer Used by the IR Library**




The size of each region is calculated as follows.

## 14.2.1.1. Reserved Region

This region is fixed size, and is required for operation of the library.

You can get its size with the following function.

**Code 14-3 Function for Getting the Size of the Reserved Region**

```
size_t nn::ir::Communicator::GetReservedSize();
```

## 14.2.1.2. Send and Receive Packet Management Region

This region is used for storing management information for transmitted packets.

Its size is calculated based on the maximum number of packets that can be maintained at any time. **For connection processing, specify a size returned by `GetManagementSize(1)` for both send and receive.**

Get the management information size using the following function.

**Code 14-4. Function to Get the Packet Management Information Size**

```
size_t nn::ir::Communicator::GetManagementSize(s32 dataNum);
```

Specify the maximum number of packets maintained at any time in *`dataNum`*.

## 14.2.1.3. Send and Receive Packet Save Region

This region is used to store sent and received packets. **For connection processing, the size must be at least 32 bytes for both send and receive.**

If the data sizes are variable, specify a size equal to or greater than that calculated using the maximum packet size.

If the data sizes are fixed, specify the size of a single data packet multiplied by the maximum number of packets.

Calculate the packet size from the data size using the following function.

**Code 14-5. Function to Get the Size of a Packet**

```
size_t nn::ir::Communicator::GetPacketSize(size_t dataSize);
```

Specify the size of the data in *dataSize*.

### Notes About the Receive Packet Save Region

Due to noise, packets stored in the received packet storage area could be invalid. Because of this, **we recommend allocating a larger area than the actual number of packets that will be received** to ensure that normal packets received after such an invalid packet will be saved reliably. For example, one approach would be to allocate the unused region to the receive packet save region after deciding the sizes for the other regions.

The following code example sends and receives fixed-size data and maximizes the allocation of the receive packet save region.

**Code 14-6. Code Sample for Calculating Region Sizes**

```
// Buffer, send/receive data sizes, and number of packets to save.
static u8 buffer[4096] NN_ATRRIBUTE_ALIGN(4096);
size_t sendDataSize = 100;
size_t sendPacketNum = 10;
size_t recvDataSize = 50;
size_t recvPacketNum = 20;

// Calculate required sizes.
size_t sendBufferSize =
        nn::ir::Communicator::GetPacketSize(sendDataSize) * sendPacketNum;
size_t sendManagementSize =
        nn::ir::Communicator::GetManagementSize(sendPacketNum);
size_t receiveBufferSize =
        nn::ir::Communicator::GetPacketSize(recvDataSize) * recvPacketNum;
size_t receiveManagementSize =
        nn::ir::Communicator::GetManagementSize(recvPacketNum);
size_t reservedSize = nn::ir::Communicator::GetReservedSize();

// Check whether the total region size is less than the buffer size.
NN_ASSERT((sendBufferSize + sendManagementSize + receiveBufferSize +
           receiveManagementSize + reservedSize) <= sizeof(buffer));

// Add an unused region size to the size of the receive packet save region.
receiveBufferSize = sizeof(buffer) -
        (sendManagementSize + sendBufferSize +
         receiveManagementSize + reservedSize);
```

# 14.3. Connections

To start infrared communications between 3DS systems, the systems must authenticate each other as communication devices. The called functions are divided into those for the system waiting for a connection request, and those for the system sending a connection request. The waiting system must call the `WaitConnection()` function, and the sending system must call the `RequireConnection()` function. Both of these functions are asynchronous calls, so control returns before connection processing has completed.

**Code 14-7. Connection Processing Functions**

```
static nn::Result nn::ir::Communicator::WaitConnection();
static nn::Result nn::ir::Communicator::WaitConnection( nn::fnd::TimeSpan
sendReplyDelay);
static nn::Result nn::ir::Communicator::RequireConnection();
static nn::ir::ConnectionStatus nn::ir::Communicator::GetConnectionStatus();
static nn::ir::TryingToConnectStatus
nn::ir::Communicator::GetTryingToConnectStatus();
```

Processing is asynchronous, so the `GetConnectionStatus()` function is provided to confirm the overall state of the library and whether connection processing has completed. It returns `CONNECTION_STATUS_CONNECTED` if connection processing has completed. The `GetTryingToConnectStatus()` function provides processing state details. For other possible return values, see 14.7. Getting the Connection.

The following table shows a list of return values for the `GetTryingToConnectStatus()` function.

**Table 14-2. GetTryingToConnectStatus Function Return Values**

| Return Value | Description |
|---|---|
| TRYING_TO_CONNECT_STATUS_NONE | Not processing a connection. |
| TRYING_TO_CONNECT_STATUS_SENDING_REQUEST | Sending a connection request. |
| TRYING_TO_CONNECT_STATUS_WAITING_REPLY | Waiting for a reply to a connection request. |
| TRYING_TO_CONNECT_STATUS_WAITING_REQUEST | Waiting for a connection request. |
| TRYING_TO_CONNECT_STATUS_SENDING_REPLY | Sending a reply to a connection request. |

## 14.3.1. Connecting Automatically

When it is not clear which of two 3DS systems to use to receive a connection request, such as when IR communications is requested from the menu on both systems at the same time, use automatic connection.

With automatic connection, both systems switch automatically between waiting for a connection and requesting a connection with different timing, in order to connect. As a result, the connection relationship changes based on the combination when it occurs.

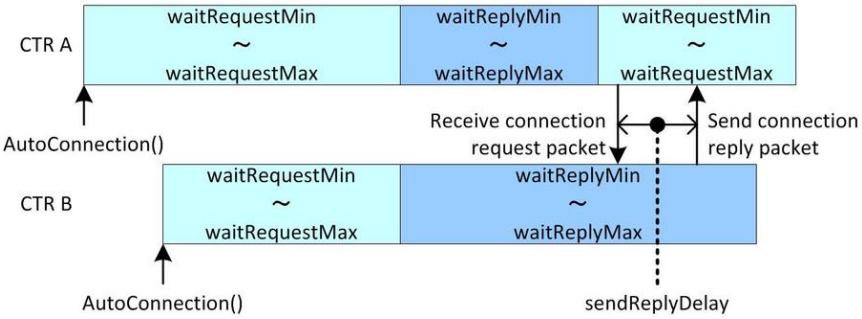
**Code 14-8. Automatic Connection Functions**

```
static nn::Result nn::ir::Communicator::AutoConnection();
static nn::Result nn::ir::Communicator::AutoConnection(
        nn::fnd::TimeSpan sendReplyDelay,
        nn::fnd::TimeSpan waitRequestMin, nn::fnd::TimeSpan waitRequestMax,
```

```
            nn::fnd::TimeSpan waitReplyMin, nn::fnd::TimeSpan waitReplyMax);
```

The following figure shows an example of the operation of automatic connection.

**Figure 14-3. Operation of Automatic Connection**




If called without parameters, values to be used are calculated by the library based on the baud rate set at initialization. Automatic connection may not succeed in some cases, when it is called without specifying parameters, due to other processing by the application. In such cases, specify the parameters while considering the following points.

*sendReplyDelay* is the interval between receiving a connection request packet and sending a connection reply packet. The default value is 3 ms. The value must be at least 3 ms, to allow the partner to switch between sending and receiving.

*waitRequestMin* and *waitRequestMax* are the minimum and maximum times for sending a connection request packet and waiting to receive a connection reply packet. *waitRequestMin* must be larger than the total of the connection request packet send time, the connection reply packet receive time, and the time for switching between sending and receiving (3 ms).

*waitReplyMin* and *waitReplyMax* are the minimum and maximum times to wait for a connection request packet and to send a connection reply packet. *waitReplyMin* must be greater than the total of the time to receive a connection request packet, the time to send a connection reply packet, and the interval for switching between sending and receiving (3 ms).

The size of a connection request packet is defined by `CONNECTION_REQUEST_PACKET_SIZE` (8 bytes), and the size of a connection reply packet is defined by `CONNECTION_REPLY_PACKET_SIZE` (5 bytes). The time required to send or receive can be calculated from the actual data transmission speed. Packet sizes are defined in bytes. Use care when calculating from transmission speeds, which are in bits.

## 14.3.2. Connection Roles

You can obtain connection roles using the `GetConnectionRole()` function. Operation of the IR library does not differ based on connection role. Use them as the policy that determines the role within infrared communication.

**Code 14-9. Function for Getting a Connection Role**

```
static nn::ir::ConnectionRole nn::ir::Communicator::GetConnectionRole(void);
```

The following table shows the return values for the `GetConnectionRole()` function.

**Table 14-3. GetConnectionRole Function Return Values**

| Return Value | Description |
|---|---|
| CONNECTION_ROLE_NONE | The connection role is not set because connection processing has not started or is in progress. |
| CONNECTION_ROLE_TO_REQUIRE | Connected after requesting a connection.<br>The system has received a connection reply packet after a RequireConnection or AutoConnection() function call. |
| CONNECTION_ROLE_TO_WAIT | Connected after receiving a connection request.<br>The system has sent a connection reply packet after a WaitConnection or AutoConnection function call. |

## 14.4. Confirming the Communication ID

To prevent communication with any other application, each system must confirm the communication ID of the other system after connection processing has completed. Any send or receive-related functions that are called before confirming communication IDs results in an error.

Use the CreateCommunicationId() function to get a communication ID to use for confirmation.

**Code 14-10. Function for Getting a Communication ID**

```
static bit32 nn::ir::Communicator::CreateCommunicationId(
        bit32 uniqueId, bool isDemo = false);
```

Specify the unique ID allocated by Nintendo for each title for *uniqueId*. When communicating between different titles, specify the unique ID for either one of them.

To prevent infrared communication between the retail version and the downloaded demo version of a title, specify true for *isDemo* in the demo version. Always set this parameter to false in the retail version, regardless of whether you want to communicate with demo versions.

**Note:** For test programs and titles that use infrared communication and have not been allocated a unique ID by Nintendo, specify a prototype software code in the range from 0xFF000 to 0xFF3FF for *uniqueId*. In retail software, you must always specify the unique ID assigned by Nintendo.

To confirm communication IDs, one 3DS calls the RequireToConfirmId() function and the other calls the WaitToConfirmId() function. Both of these functions complete successfully if the communication IDs, communication mode IDs, and passphrases specified by each system match. Send and receive functions can only be used after this point.

**Code 14-11. Functions for Confirming Communication IDs**

```
static nn::Result nn::ir::Communicator::RequireToConfirmId(
        u8 subId, bit32 communicationId,
        const char passphrase[], size_t passphraseLength);
static nn::Result nn::ir::Communicator::WaitToConfirmId(
        u8 subId, bit32 communicationId,
        const char passphrase[], size_t passphraseLength,
        nn::fnd::TimeSpan timeout);
static bool nn::ir::Communicator::IsConfirmedId();
```

Use the communication ID generated by `CreateCommunicationId` for *`communicationId`*.

When an application using infrared communication has multiple scenes, communication between different scenes can be prevented by specifying a communication mode identifier in *`subId`*. Specify a different value for each scene.

The passphrase specified in *`passphrase`* is used as the key for encrypting infrared communications packets. Avoid using strings that are easy to guess. The length of a passphrase must be in the range between `IR_PASSPHRASE_LENGTH_MIN` and `IR_PASSPHRASE_LENGTH_MAX`.

Specify the length of the passphrase in *`passphraseLength`*.

Note that both the `RequireToConfirmId()` and `WaitToConfirmID()` functions implicitly consume one of the `nn::os::Event` class instances assigned to the application.

Communication IDs must be reconfirmed after a disconnect and reconnection, even if they have been confirmed earlier.

The `IsConfirmedId()` function can be used to check whether ID confirmation has already been completed.

## 14.5. Transmission

Data can be sent by infrared communications using the `Send()` function after completing confirmation of communication IDs.

**Code 14-12. Functions for Transmitting Data**

```
static nn::Result nn::ir::Communicator::Send(
        void *pBuffer, size_t dataSize, size_t bufferSize,
        bool restore = false);
static void nn::ir::Communicator::GetSendEvent(nn::os::Event* pEvent);
static nn::Result nn::ir::Communicator::GetLatestSendErrorResult(bool clear);
static void nn::ir::Communicator::GetSendSizeFreeAndUsed(
        size_t *pSizeFree, s32 *pCountFree,
        size_t *pSizeUsed, s32 *pCountUsed);
```

Specify the send buffer and its size in *`pBuffer`* and *`bufferSize`* respectively. The send buffer must be at least the size obtained by passing the size of the raw data to be sent to the `CalculateBufferSizeToCommunicate()` function. In addition, this buffer must have a starting address that is `nn::ir::Communicator::SEND_BUFFER_ALIGNMENT` (4 bytes) aligned.

Specify the data size in *`dataSize`*. Data in the range from 0 to 16316 bytes can be sent.

The data starting at the beginning of *`pBuffer`* and of length *`dataSize`* is sent. After data is sent, the buffer contains the encrypted data with headers appended. If the original data is needed after it is sent, specify `true` for the *`restore`* parameter. Note that this consumes more processing time than only sending.

The actual transmission is done asynchronously and the `Send()` function returns control to the caller after the packet has been saved in the buffer being used by the library. An event notifies when the data is actually sent, and can be obtained using the `GetSendEvent()` function. The event passed to *`pEvent`* is initialized as an automatically resetting event. The `Result` class instance returned by the `GetLatestSendErrorResult()` function indicates whether a send error occurred. If called with the *`clear`* parameter set to `true`, the error in the library's internal `Result` class instance is cleared.

The library monitors the state of the buffer, and transmits send packets as quickly as possible through the IR communications module. If the send function is called repeatedly in a short period of

time, the library could temporarily store multiple packets in the buffer at the same time. These packets are sent in the order they were saved in the buffer.

If the total size of packets saved exceeds the size of the buffer when sending large packets or sending packets repeatedly, the library discards any further packets without saving them. For this reason, we recommend using `GetSendSizeFreeAndUsed` to check the available space in the send packet management region and the save region before sending a packet. In particular, note that available space in the save region of at least the packet size obtained from the `GetPacketSize()` function is required in order to save the packet.

## 14.6. Receipt

Data can be received by calling the `Receive()` function after completing confirmation of the communications IDs. After the connection process, the library can continue to receive data unless it is maintaining a packet to be sent. So strictly speaking, the `Receive()` function is not a function that performs receipt. It retrieves data that has already been received from the buffer. It is not possible to determine whether received data is valid until it has been retrieved using the `Receive()` function.

**Code 14-13. Functions Using Receive Processes**

```
static nn::Result nn::ir::Communicator::Receive(
        void* pDst, size_t size, size_t *pReceiveSize,
        s32 *pRemainCount);
static void nn::ir::Communicator::GetReceiveEvent(nn::os::Event* pEvent);
static nn::Result nn::ir::Communicator::GetLatestReceiveErrorResult(
        bool clear);
static nn::Result nn::ir::Communicator::GetNextReceiveDataSize(size_t *pSize);
static nn::Result nn::ir::Communicator::DropNextReceiveData(s32 *pRemainCount);
static void nn::ir::Communicator::GetReceiveSizeFreeAndUsed(
        size_t *pSizeFree, s32 *pCountFree,
        size_t *pSizeUsed, s32 *pCountUsed);
```

Specify the receive buffer and buffer size in *`pDst`* and *`size`* respectively. The receive buffer's starting address must be `nn::ir::Communicator::RECEIVE_BUFFER_ALIGNMENT` (4 bytes) aligned. The data is temporarily saved in encrypted form with appended headers (and so forth) in the buffer, so the buffer must be at least as large as the size calculated by the `CalculateBufferSizeToCommunicate()` function (the size of the raw data to be received) when passed. The required buffer size can be retrieved beforehand using the `GetNextReceiveDataSize()` function.

One packet of data is retrieved for each call to the `Receive()` function. The size of the retrieved data is stored in *`pReceiveSize`*. When storing the data, the `Receive()` function stores the amount remaining to be retrieved in *`pRemainCount`*. To read all of the stored data, call the `Receive()` function repeatedly until 0 (zero) bytes are remaining.

The `GetReceiveEvent()` function returns an event containing notification of when received packets were saved in the buffer, so it can be used if data must be retrieved as soon as it has been received. The event passed to *`pEvent`* is initialized as an automatically resetting event. The `GetLatestReceiveErrorResult()` function returns an instance of the `Result` class, indicating whether an error has occurred during receive processing. If called with the *`clear`* parameter set to `true`, the error in the library's internal `Result` class instance is cleared.

If the `GetNextReceiveDataSize()` function returns a data size that is clearly different from the expected data (such that the data in the next packet is not needed), one packet's worth of data can be discarded by using the `DropNextReceiveData()` function.

If packets are received more frequently than data is retrieved and the total size of packets to be saved exceeds the size of the buffer, the library discards any further packets without saving them. Use the `GetReceiveSizeFreeAndUsed` function to determine the available space for the receive packet management region and the save region, and use it to manage receipt.

If the current connection is interrupted while received data remains, the remaining data can be received in the interval until the next connection begins.

# 14.7. Getting the Connection Status

Use the `GetConnectionStatus()` function to get the status of the connection. In addition, the `GetConnectionStatusEvent()` function returns an event that indicates when the connection status changed. The event passed to *`pEvent`* is initialized as an automatically resetting event.

**Code 14-14. Functions Used to Get the Connection Status**

```
static void nn::ir::Communicator::GetConnectionStatusEvent(
        nn::os::Event* pEvent);
static nn::ir::ConnectionStatus nn::ir::Communicator::GetConnectionStatus();
```

The following table shows the return values from the `GetConnectionStatus()` function.

**Table 14-4. GetConnectionStatus Return Values**

| Return Value | Description |
|---|---|
| `CONNECTION_STATUS_STOPPED` | The infrared module is stopped. |
| `CONNECTION_STATUS_TRYING_TO_CONNECT` | Indicates that a connection is being established. |
| `CONNECTION_STATUS_CONNECTED` | A connection has been established. |
| `CONNECTION_STATUS_DISCONNECTING` | Indicates that a connection is being disconnected. |
| `CONNECTION_STATUS_FATAL_ERROR` | A fault or other fatal error has occurred. |

## 14.7.1. Handling Faults

When the connection status is `CONNECTION_STATUS_FATAL_ERROR`, a fault may have occurred in the infrared communication module. In this state, all functions that return a result except for `Finalize` always `return nn::ir::ResultFatalError`. Infrared communication cannot be used in any further processing. Immediately finalize the IR library.

This state could indicate a fault in the infrared module, but power-cycling the system may also recover the module. We recommend displaying a message to the user indicating the possibility of a fault and how to handle it.

# 14.8. Disconnecting

To disconnect infrared communication, call the `Disconnect()` function.

**Code 14-15. Functions Using the Disconnect Processes**

```
static nn::Result nn::ir::Communicator::Disconnect(void);
static nn::Result nn::ir::Communicator::ClearSendBuffer(void);
static nn::Result nn::ir::Communicator::ClearReceiveBuffer(void);
```

The `Disconnect()` function sends a disconnect request to the communication partner. If a connection is being established, the process is interrupted. Disconnection is handled asynchronously, so control is returned before disconnection processing has completed.

If `Disconnect` is called while some send packets are still unprocessed, the send packets are discarded and the connection is disconnected. If necessary, confirm that all packets have been sent before calling this function. To explicitly discard packets not sent, call the `ClearSendBuffer()` function.

Packets received before disconnection can be retrieved using the `Receive()` function after disconnecting. Note, however, that they will be discarded when processing the next connection and connection authentication begins. To explicitly discard all received packets, call the `ClearReceiveBuffer()` function.

## 14.9. Finalization

To finalize use of the IR library, call the `Finalize()` function.

**Code 14-16. Functions Using the Finalization Processes**

```
static nn::Result nn::ir::Communicator::Finalize(void);
```

If this function is called before the library is initialized, the `nn::ir::ResultNotInitialized()` function returns an error.



# 15. Differences Between TWL Mode and Actual TWL Systems

An application works differently on a 3DS system in TWL mode (DSi-compatible mode) than it does on an actual Nintendo DSi system. This chapter describes those differences. Consider these differences when you are developing applications for systems in the Nintendo DS family that will also run on 3DS systems. (In particular, refer to this information when developing a Nintendo DSiWare service or game that is available on Nintendo eShop.)

**Note:** For information about how to import a Nintendo DSiWare service or game, see the API

Reference page for `ctr_makecia`.

# 15.1. Display-Related Differences

## 15.1.1. LCD Screens

Because the LCD screen layouts differ (see 5.5 Initializing the GX Library), the direction of scanning on the 3DS screens is different from that on the TWL screens. Also note that because emulation is used, the images are rendered with a delay of approximately 1.3 frames.

# 15.2. Input-Related Differences

## 15.2.1. Touch Panel

Normally, on TWL or NITRO systems, it is difficult for applications to detect touches at the very edge of the screen. But when the application has been started in 1:1 pixel display mode on the 3DS, the SDK (TWL or NITRO) functions might be able to detect touches at the very edges of the application's screen (a display area that is a subset of the physical screen) because the user has touched the screen outside of the application's display area.

## 15.2.2. Key Input

When the +Control Pad is being emulated by the Circle Pad (see 6.2.1. Digital Buttons and Circle Pad), although the state of the +Control Pad is updated in real time when the Circle Pad is not being moved (when no input is detected), it updates roughly once per frame when the Circle Pad is being moved. For this reason, input from the +Control Pad may not always be reflected accurately when the user is operating the Circle Pad.

When the Circle Pad and the +Control Pad are both being used and the user simultaneously presses Up and Down (or Left and Right) on the +Control Pad, the application is notified of a prohibited operation. The input is treated as Up when Up and Down are pressed simultaneously, and Left when Left and Right are pressed simultaneously. If your application uses such prohibited input to trigger entry to debug mode or some other mode, your application will not operate correctly, because it will never receive the prohibited input.

## 15.2.3. Closing the System

For the application, the system state while displaying the HOME Menu is no different from when the system is closed. The `PAD_DetectFold()` function returns TRUE just like it normally does when the system is closed, but the application is not notified of key input or touch panel input.

### 15.2.4. Sound Volume

The system has a slider for sound volume. This allows users to change the volume faster than is physically possible on the Nintendo DSi.

## 15.3. OS-Related Differences

### 15.3.1. Reset and Shutdown

On CTR, processing of the `OS_RebootSystem()` function takes much longer than it does on a TWL system.

### 15.3.2. Application Jumps

On CTR, the filename and content of the system menu version are different from TW. If your application incorporates routines that check this information, it might operate incorrectly when running in TWL mode on a CTR system.

## 15.4. System Settings-Related Differences

### 15.4.1. Region Codes

On 3DS, the Australia region is included in the Europe region. Applications intended for the TWL Australia region can also operate in the 3DS Europe region.

**Table 15-1. TWL Application Regions and Corresponding Operable 3DS System Regions**

| TWL Application Region | Operable 3DS System Regions |
|---|---|
| JP (Japan) | JP (Japan) |
| US (United States) | US (United States) |
| EU (Europe) | EU (Europe) |

| | |
|---|---|
| AU (Australia) | EU (Europe) |
| EU (Europe) and US (United States) | EU (Europe) or US (United States) |
| EU (Europe) and AU (Australia) | EU (Europe) |
| EU (Europe), AU (Australia), and US (United States) | EU (Europe) or US (United States) |
| CN (China) | CN (China) |
| KR (Korea) | KR (Korea) |

### 15.4.2. Country Setting

If the country setting of the 3DS system is set to a value that would not be available in the region of the TWL application, the value of the `country` member of the `OSOwnerInfoEx` structure is set to `254 (OTHER)`.

### 15.4.3. Language Setting

If the language setting of the 3DS system is set to a value that would not be available in the region of the TWL application, the `language` member of the `OSOwnerInfoEx` structure is set to either `OS_LANGUAGE_JAPANESE` for the Japan region or to `OS_LANGUAGE_ENGLISH` for all other regions.



# 16. Appendix: Process Flows for Using the Circle Pad Pro

Because status indicators are not built into the Circle Pad Pro, it is unable to determine from the outside whether it is active, whether its battery level is low, or whether it is connected to the CTR system. To ensure ease of use, some mechanism is required for the detection, connection, and redetection processes of the Circle Pad Pro.

> **Note:** Because the C Stick, ZL Button, and ZR Button are included in the SNAKE hardware, the system is treated as having a Circle Pad Pro always attached. The remaining battery life also never decreases.
>
> Consequently, your application can omit some of the process flows described in this section if you verify the application is running on SNAKE hardware in advance.

This chapter provides Nintendo's recommended process flows. Use them as a reference upon implementation. Yellow backgrounds indicate an example of the message to be displayed for that process.

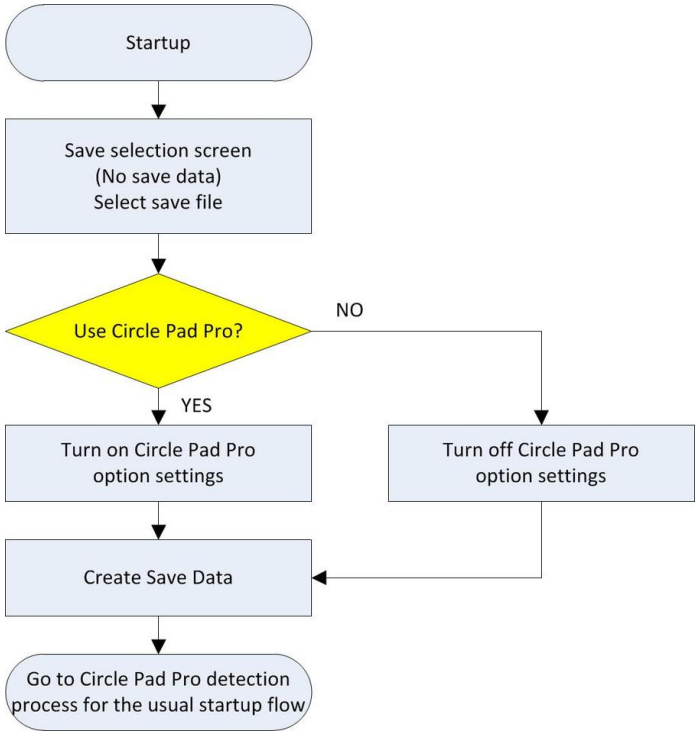
Refer to the following when deciding whether to use "Circle Pad Pro" or "C Stick" in user-facing messages.

- Display as "Circle Pad Pro"
  - CTR Application
  - When the Circle Pad Pro is used on CTR and the C Stick on SNAKE in a SNAKE-compatible title
- Display as "C Stick"
  - SNAKE-Only Title
  - When the C Stick is used on SNAKE but the Circle Pad Pro is not used on CTR in a SNAKE-compatible title

## 16.1. Process Flow for Initial Startup

To allow the save data to be saved when using the Circle Pad Pro, we recommend conforming to the following process flow upon initial startup.

**Figure 16-1. Process Flow for Initial Startup**

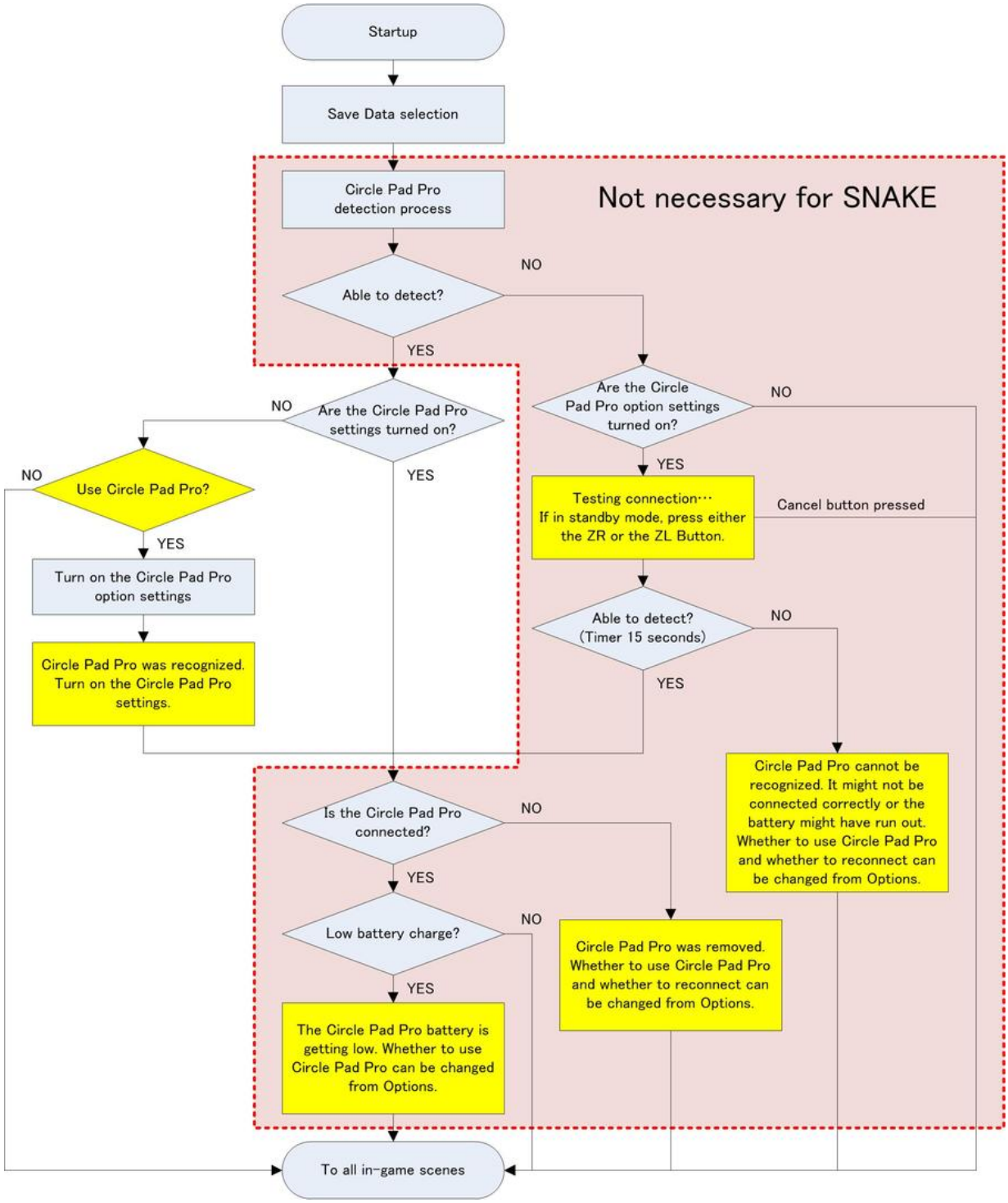



## 16.2. Process Flow for Normal-Usage Startup

As shown in 6.1. Process Flow for Initial Startup, to allow the save data to be saved using the Circle Pad Pro, we recommend conforming to the following process flow upon normal-usage startup.

Parts of the flow that are not necessary on SNAKE are indicated with a dotted red line around a light red background.

**Figure 16-2. Process Flow for Normal-Usage Startup**

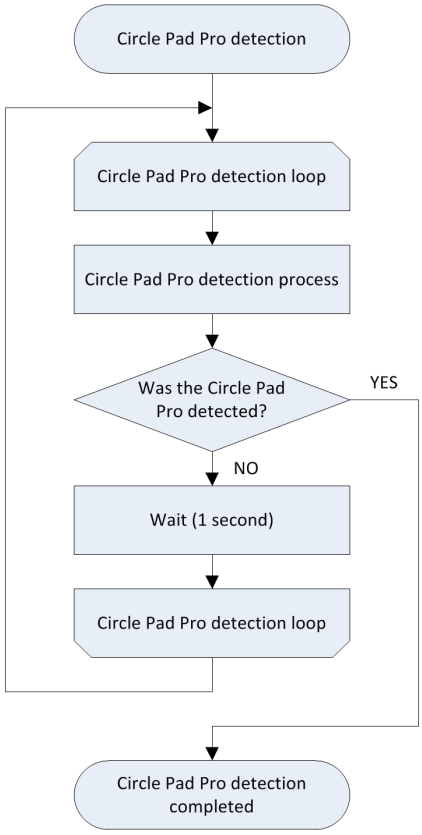

Startup
Save Data selection
Circle Pad Pro detection process
Not necessary for SNAKE
Able to detect?
NO
YES
Are the Circle Pad Pro settings turned on?
NO
YES
Use Circle Pad Pro?
NO
YES
Turn on the Circle Pad Pro option settings
Circle Pad Pro was recognized. Turn on the Circle Pad Pro settings.
Are the Circle Pad Pro option settings turned on?
NO
YES
Testing connection… If in standby mode, press either the ZR or the ZL Button.
Cancel button pressed
Able to detect? (Timer 15 seconds)
NO
YES
Circle Pad Pro cannot be recognized. It might not be connected correctly or the battery might have run out. Whether to use Circle Pad Pro and whether to reconnect can be changed from Options.
Is the Circle Pad Pro connected?
NO
YES
Low battery charge?
NO
YES
Circle Pad Pro was removed. Whether to use Circle Pad Pro and whether to reconnect can be changed from Options.
The Circle Pad Pro battery is getting low. Whether to use Circle Pad Pro can be changed from Options.
To all in-game scenes

## 16.3. Process Flow for Detecting the Circle Pad Pro

**Note:** The following process flow does not apply to SNAKE.

We recommend the following process flow for detecting the Circle Pad Pro, except for when it is detected after initial startup. The following figure assumes that the Circle Pad Pro is detected from a thread other than the main thread. To detect explicitly, or to set up an option for connection, refer to 16.4. Options Screen Process Flows.

**Figure 16-3. Process Flow for Detecting the Circle Pad Pro**



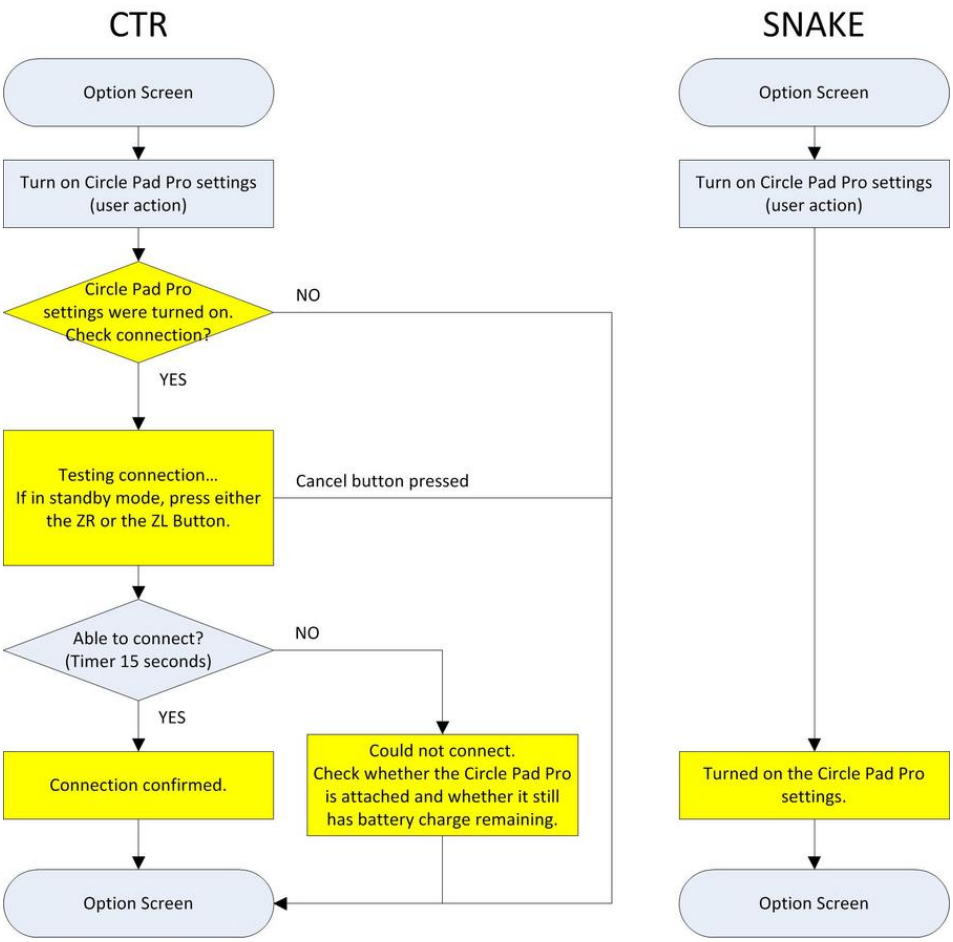

# 16.4. Options Screen Process Flows

When using the Options screen to enable the Circle Pad Pro and transition to connection/reconnection, we recommend the following process flows.

## 16.4.1. Transition to Enabling the Circle Pad Pro

Using the Options screen to transition to enabling the Circle Pad Pro, we recommend the following process flow.

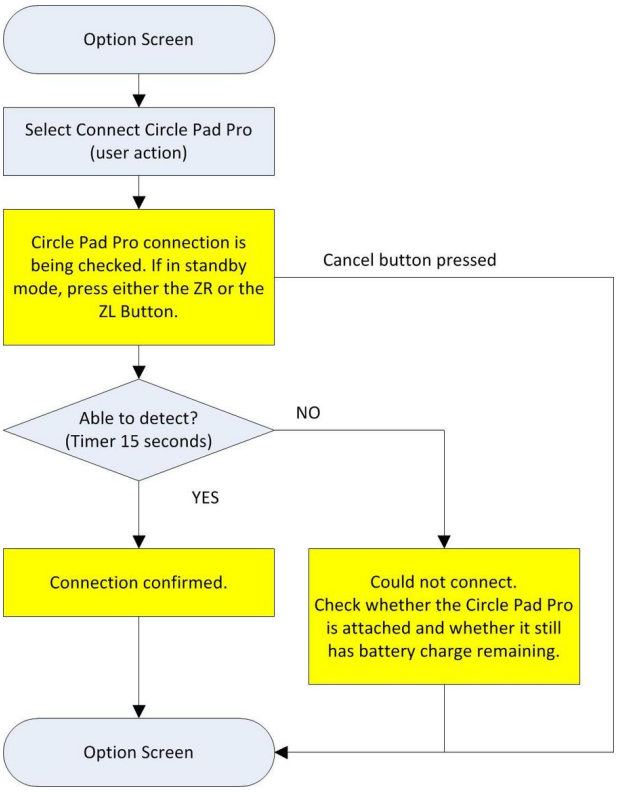
**Figure 16-4. Process Flow for Enabling the Circle Pad Pro**


CTR
SNAKE
Option Screen
Turn on Circle Pad Pro settings (user action)
Circle Pad Pro settings were turned on. Check connection?
NO
YES
Testing connection... If in standby mode, press either the ZR or the ZL Button.
Cancel button pressed
Able to connect? (Timer 15 seconds)
NO
YES
Could not connect. Check whether the Circle Pad Pro is attached and whether it still has battery charge remaining.
Connection confirmed.
Option Screen
Option Screen
Turn on Circle Pad Pro settings (user action)
Turned on the Circle Pad Pro settings.
Option Screen

**Note:** On SNAKE, the system is always connected when sampling begins, so the following process flow is unnecessary.

When setting up the option to connect to the Circle Pad Pro, use the following process flow.

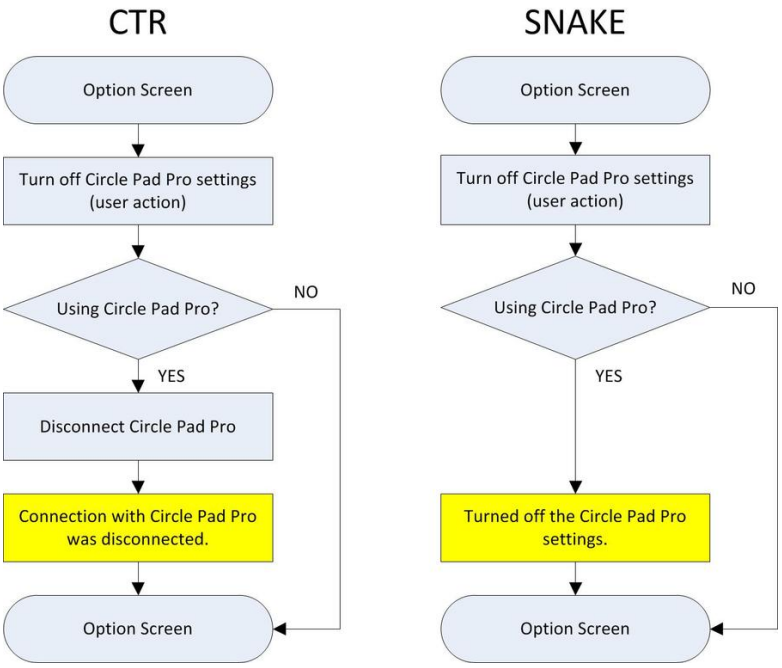
**Figure 16-5 Process Flow for Connecting the Circle Pad Pro**


Option Screen
Select Connect Circle Pad Pro (user action)
Circle Pad Pro connection is being checked. If in standby mode, press either the ZR or the ZL Button.
Cancel button pressed
Able to detect? (Timer 15 seconds)
NO
YES
Connection confirmed.
Could not connect. Check whether the Circle Pad Pro is attached and whether it still has battery charge remaining.
Option Screen

## 16.4.2. Process Flow for Disconnecting the Circle Pad Pro

When setting up the option to disconnect from the Circle Pad Pro, we recommend the following process flow.

**Figure 16-6. Process Flow for Disconnecting the Circle Pad Pro**



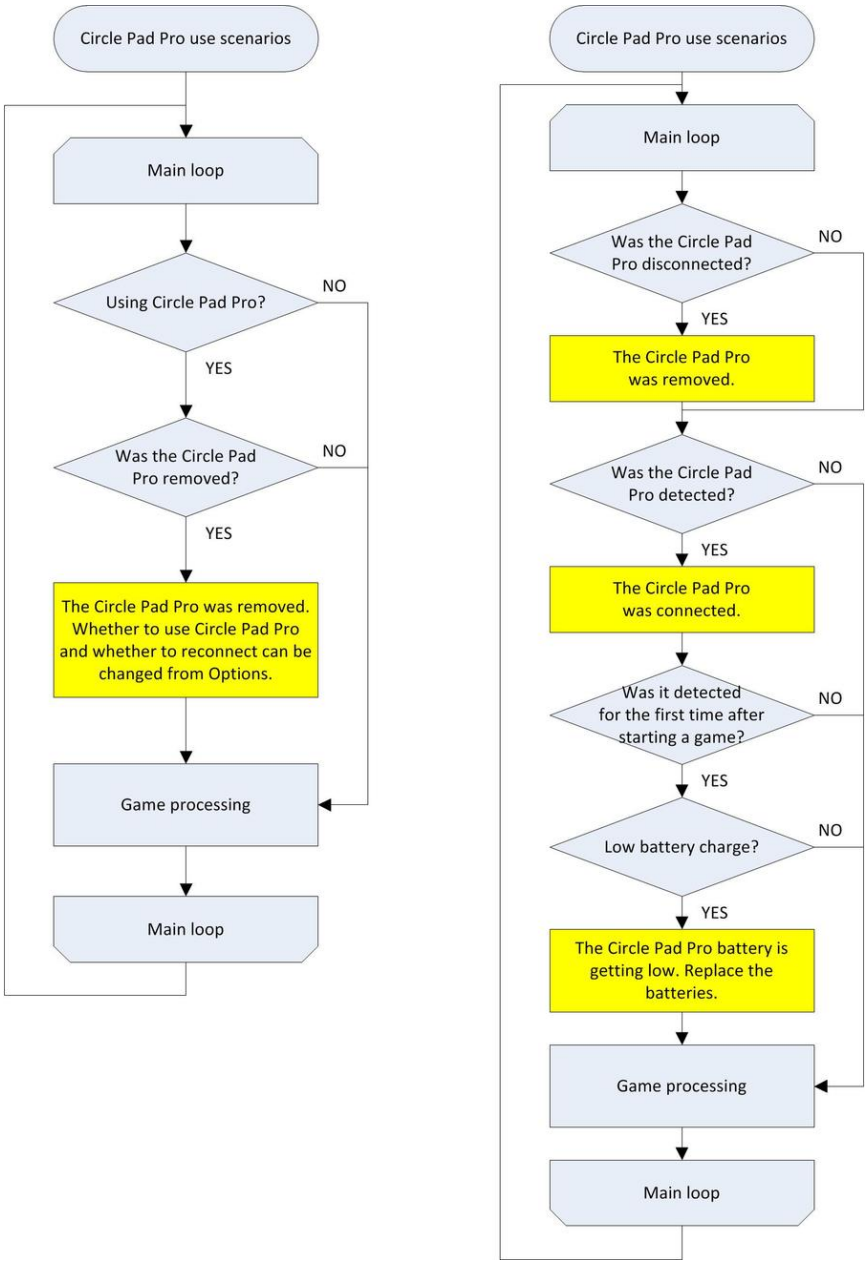
# 16.5. Process Flow for Using the Circle Pad Pro

**Note:** The following process flow does not apply to SNAKE.

We recommend the following process flow for using the Circle Pad Pro.

In the figures below, the process is determined by either setting up the option for the Circle Pad Pro detection in the application or by monitoring full-time, before being incorporated into the main thread. The figure on the left shows cases when the option for redetecting the Circle Pad Pro is set up in the application, and the figure on the right shows cases when the Circle Pad Pro is being pulsed for detection full-time. Refer to Figure 16-3. Process Flow for Detecting the Circle Pad Pro for cases when the Circle Pad Pro is being monitored full-time in the thread other than the main thread.

**Figure 16-7. Process Flow for Circle Pad Pro Detection in Application (left) or Full-Time (right)**



# 17. Appendix: Testing the Operations of Standard Applications on SNAKE

SNAKE has new features, including Super-Stable 3D, an additional input device, and better camera performance. To realize these features, the SNAKE hardware is partially incompatible with CTR. System processes also differ. For more information about SNAKE's added features, see the *3DS Overview*.

There may be differences in behavior when running standard applications on CTR and when running them on SNAKE, so **please test standard applications on both CTR and SNAKE**.

When testing your titles, note the following.

| | |
|---|---|
| Performance-pursuing scenes | SNAKE has a higher memory access latency than the CTR, so waiting for memory access could cause some application processes to take longer. Application processing has been found to take as much as around 5% longer in some scenes. If your application has been tuned to the limits on the CTR, there is a chance of processing slowdowns on SNAKE.<br><br>You can handle this in your application in two ways: lighten the processing burden, or build the application as an extended application that runs three times faster (at 804 MHz) on SNAKE. For more information, see 4.2. Standard and Extended Applications. |
| Local communication | SNAKE processes local communications differently from the CTR, so unexpected behaviors are possible when local communications take place among a mix of the two systems.<br>To verify the operations of local communications, use a mix of CTR and SNAKE systems and test operations with each acting as the host. |
| Starting and ending the cameras | The Super-Stable 3D feature of SNAKE uses input from the cameras, so this feature is disabled when the application uses the cameras.<br>Note that on SNAKE, initializing and finalizing the camera library involves the processes to disable and enable the Super-Stable 3D feature, so it takes a little longer to initialize and finalize the library. |
| HOME Menu | On SNAKE, the HOME Menu operates in extended mode. When a standard application is started, the mode switches to standard mode.<br>Strictly speaking, the standard application runs in extended mode until `nn::applet::Enable()` is called, at which time the application switches to standard mode. For more information, see 4.2.4. When the Operating Mode Changes.<br><br>Note that when the HOME Button is pressed, the entire system including the suspended application operates in extended mode. |
| Input interface differences | • The inner camera on SNAKE has a wider angle than the inner camera on CTR systems.<br>• The inner and outer cameras on SNAKE have less noise than those on the CTR systems.<br>• SNAKE has an automatic LCD brightness adjustment function. (Because the auto-brightness adjustment feature uses camera input, it is disabled when an application is using the camera; the immediately prior screen brightness level is maintained.)<br>• Using the C Stick on SNAKE has a different feel than the Circle Pad on the Circle Pad Pro.<br>• The C Stick on SNAKE and the Circle Pad on the Circle Pad Pro change their state differently upon waking up from Sleep. For more information, see 6.2.7.2. Differences From the Circle Pad Pro.<br>• The SNAKE NFC has a different feel than the NFC Reader/Writer. |

# 18. Appendix: Important Notes on 3DS Application Development Using the IS-SNAKE DevKit

Although the IS-SNAKE DevKit is a tool for developing SNAKE applications, it can also be used for developing standard applications for CTR.

As noted in 4.2. Standard and Extended Applications, standard applications run in standard mode for both CTR and SNAKE. However, when extended applications are run on CTR and SNAKE, the CPU's running speed and usable memory size are different. Extended applications are normally run in extended mode in the IS-SNAKE DevKit, but by using forced CTR-compatibility mode, you can test in standard mode even for extended applications.

However, as shown in the table in 17. Appendix: Testing the Operations of Standard Applications on SNAKE, operations in the IS-SNAKE DevKit may differ from behavior on CTR because some of the specifications are incompatible. For both standard applications and extended applications, always perform testing in the CTR environment.

**Notes:**For more information about forced CTR-compatibility mode, see *IS-CTR-DEBUGGER Help*.

CONFIDENTIAL

# Revision History

## Version 1.6 (2016-06-24)

### Changes

- 17. Appendix: Testing the Operations of Standard Applications on SNAKE
    - Added that the auto-brightness adjustment feature is disabled during camera use.

## Version 1.5 (2016-05-10)

### Additions

- 18. Appendix: Important Notes on 3DS Application Development Using the IS-SNAKE DevKit

### Changes

- 3.1.2. Device Memory
    - Added conditions to use when resizing device memory.
- 3.1.4. Heap Memory
    - Added conditions to use when resizing heap memory.
- 17. Appendix: Testing the Operations of Standard Applications on SNAKE
    - Moved operations testing with the IS-SNAKE DevKit to a separate page.

## Version 1.4 (2015-11-05)

### Additions

- 12.1.9. Login Applet

### Changes

- Overall
    - Appended a definition name for the alignment restriction.
- 4.3.2. Libraries
    - Added `aacdec`, `aacenc`, `act`, `nfp`, and `qtm` to the library list.
- 6.2.6. Circle Pad Pro
    - Added an instruction to end any other infrared communication functions in advance if they are being used.
- 6.2.6.3. Initialization

    - Changed the buffer size specified by the Circle Pad Pro initialization function to 12,288 bytes.

- 9.4. Scheduling
    - Added a warning describing the drop in system performance caused by excessive repeated calls of `Sleep()`.

- 12.1.1. Information Common to All Library Applets
    - Added information describing the login applet.

- 12.1.7. Circle Pad Pro Calibration Applet
    - Including `libnn_extrapad` is no longer needed with CTR-SDK 11.4 or higher.

- 14. Infrared Communication Between Systems
    - Added an instruction to end any other infrared communication functions in advance if they are being used.

## Version 1.3 (2015-04-28)

### Changes

- 2.7.1. Game Card Slot
    - Revised the description of which cards can be inserted in the Game Card slot.

- 2.9.1. Speakers
    - Added information about the system speaker sound pressure frequency characteristics.

- 5.3.2. Handling HOME Button Presses
    - Added the NFC to the descriptions for handling devices during HOME Menu display.

- 5.3.9. Jump to Nintendo eShop
    - Added information about jumping to the patch page.

- 7.1.8. Latency Emulation
    - Fixed the hierarchy of the Config tools menu to match the current tools.

- 7.3. Save Data
    - Added to the errors that require handling when mounting a save data region.
    - Added that some errors are not returned for some media.
- 7.4. Extended Save Data
    - Added to the errors that require handling when mounting an extended save data region.
- 7.6.1. Nintendo 3DS Cards
    - Deleted the descriptions if error handling relating to save data.
    - Fixed the hierarchy of the Config tools menu to match the current tools.
- 17. Appendix: Testing the Operations of CTR Titles on SNAKE
    - Added a note about the forced CTR-compatibility mode of the IS-SNAKE Devkit.
    - Added information about input interface differences.

## Version 1.2 (2015-01-15)

### Additions

- 12. Applets
    - Merged the information about library applets and system applets into a single chapter.
- 12.2. System Applets
- 12.2.2.1. Starting an Application From the Miiverse Post Page
- 17. Appendix: Testing the Operations of CTR Titles on SNAKE

### Changes

- 1. Overview
    - Revised the link descriptions according to the addition of the Applets page.
- 4.3.2. Libraries
    - Revised the description of the OLV library.

- 5.7.2. Characteristics and Limitations of the RO Library
    - Added an upper limit to the number of dynamic modules that can be loaded simultaneously.
- 6.2.7.2. Differences From the Circle Pad Pro
    - Added information about system behavior when waking from Sleep Mode.
- 12.1. Library Applets
    - Removed redundant parts from the applet description.
    - Moved the chapter.
    - Removed unnecessary library information.
- 12.2.1. Internet Browser Applet
    - Moved the chapter.
    - Changed the chapter name from WEBBRS Library (Internet Browser Applet).
    - Added the library name.
- 12.2.2. Miiverse Application and Post App
    - Changed the chapter name from "OLV Library (Miiverse Post App)."
    - Revised the descriptions pertaining to the OLV library.
    - Moved the chapter.
- 13. Supplemental Libraries
    - Moved the chapter about system applets.

### Deletions

- Platform Notation
    - Deleted this page because the information about platform notation was moved to the Readme file.

## Version 1.1 (2014-11-10)

### Additions

- 5.3.10. Jump to E-manual

- 12.2.2. Miiverse Application and Post App

### Changes

- 2.3.1. CPU
    - Added descriptions of the core configuration and use cases.
- 4.2.6. SNAKE-Only Titles
    - Added a description of SNAKE-only titles.
- 4.3.2. Libraries
    - Added a description of adding the OLV library.
- 9.6. Thread-Local Storage
    - Added a description of the feature to call the thread local storage destructor function.

## Version 1.0 (2014-09-04)

### Additions and Changes

- Initial version.

# 3DS Programming Manual: Basic Graphics

Version 1.6

**Nintendo Confidential**

This document contains confidential and proprietary information of Nintendo, and is protected under confidentiality agreements as well as the intellectual property laws of the United States and of other countries. No part of this document may be released, distributed, transmitted, or reproduced in any form , including by any electronic or mechanical means and by including within information storage and retrieval systems, without written permission from Nintendo.

---

©2015–2016 Nintendo. All rights reserved.

All company and product names in this document are the trademarks or registered trademarks of their respective companies.

CTR-06-0077-002-F

# 1. Introduction

This document explains basic information and programming procedures necessary to use 3DS graphics features. 3D graphics features for 3DS are based on OpenGL ES 1.1. To read this document, you must have an understanding of OpenGL in addition to matrix and vector math.

**Warning:** Even though the 3DS system supports the use of OpenGL, for performance reasons, it is often advisable to avoid calling GL functions when you frequently employ 3D graphics features. Instead, write directly to PICA registers using 3D commands and command caches, as described in the *3DS Programming Manual: Advanced Graphics*. Please review the *CTR Programming Manual: Advanced Graphics*, which explains many other features, after you read this document.

Four libraries are provided for drawing graphics: GL, GD, GR, and the collection of graphics libraries included in NintendoWare for CTR (NW4C), distributed by Nintendo.

GL is a basic library that provides functions conforming to OpenGL ES specifications and a rich variety of features, including error handling and state delta management. Accordingly, this library places a heavy processing load on the system. We do not recommend using the GL library when high performance is required.

GD is a lighter-weight library than GL. Although it is not compatible with the GL library, it

has equivalent features and provides an easy-to-use API. The GD library requires a certain amount of CPU cycles for internal processing.

GR is a library intended to support the direct creation of 3D commands. Although this library executes the fastest, its use requires in-depth knowledge of how registers are set, meaning that error handling is the responsibility of the developer.

NW4C provides libraries and source code files that use command cache-related features, so be sure to look into the possible use of NW4C. Although NW4C has been implemented to take fullest advantage of optimized performance when used without modification, developers who want to create customized processing can use the GR and GD libraries in combination with NW4C.

**Note:** There are alignment and size restrictions on the memory passed from the application to the graphics library. Within the programming manual you will find numerical references to the alignment and size restrictions, but they are defined by constants. For more information about the defined constants, see the API reference.

## 1.1. Document Structure

2. GPU provides an overview of the GPU built into the 3DS system, in addition to a brief summary of its features. This chapter also explains the order in which graphics are processed. Read this chapter to get a grasp of what you can render using 3DS graphics, in addition to an overall picture of graphics processing.

3. LCD explains the relationship between the GX library and the LCD screens built into the 3DS system. Read this chapter to understand the close connection between the GX library and the images displayed on the LCDs.

4. Command Lists explains command lists, which are required to execute 3D graphics commands.

5. Shader Programs describes the types of shader programs and explains how to use them.

6. Vertex Buffers explains how to create and use vertex buffers to input batches of vertex data to the vertex shader.

7. Textures describes the types of texture images that can be handled by the 3DS system, and the CTR native format.

8. Vertex Shaders describes input vertex data, and the vertex attributes that are output for use in later-stage processing.

9. Geometry Shaders provides an overview of and explains how to use the shaders provided by the SDK for generating basic geometry, such as points and lines, and for other useful features, such as silhouettes and subdivision.

10. Rasterization describes the processing that immediately precedes per-fragment operations. The 3DS system runs the scissor test at this stage.

11. Texture Processing explains how texture units, combiners, and combiner buffers determine texel colors; how to combine lights with colors; and how to create procedural textures.

12. Reserved Fragment Shaders describes per-fragment operations that can be controlled using reserved uniforms, such as fragment lighting, shadows, fog, and gas rendering.

13. Per-Fragment Operations describes the operations performed on fragments, before they are written to the render buffer. Refer to this chapter for more information about blending, and the alpha test and other tests.

14. Framebuffer Operations describes framebuffer processing and explains how to clear buffers.

15. Miscellaneous summarizes the differences between the CTR graphics libraries and the OpenGL ES specifications.

**Note:** *3DS Programming Manual: Advanced Graphics* presents PICA register information, graphics command caching, and sample implementations of lighting models. It also introduces features that require special settings such as stereoscopic display, block mode, early depth tests, and so on.

## 1.2. Graphics Processing Errors

3D graphics for 3DS are based on OpenGL ES. Behavior follows the same specifications as OpenGL ES if an error occurs during a call to a 3D graphics function.

- The processing of the function is ignored, except when there is a `GL_OUT_OF_MEMORY` error.
- You can get error codes with the `glGetError()` function.
- Only a single error code is recorded. The error code is not updated until it is read by the `glGetError()` function, even if another error is generated in the meantime.

The following error codes are recorded.

**Table 1-1. Error Code List**

| Error Code | Conditions for Generating an Error |
|---|---|
| `GL_INVALID_ENUM` | An out-of-range value was specified as a `GLenum` parameter. |
| `GL_INVALID_VALUE` | An out-of-range value was specified as a numeric argument (and similar cases). |
| `GL_INVALID_OPERATION` | A function was called when it was prohibited to do so (and similar cases). |
| `GL_OUT_OF_MEMORY` | There is not enough memory to execute the function. |

## 1.3. Single-Threaded Model

The 3D graphics library was designed under the assumption that it would be called from a single thread. Operation is not guaranteed when functions are called from multiple threads. Functions must also be called on only one rendering context.



# 2. GPU

A PICA graphics core (268 MHz) developed by Digital Media Professionals (DMP) is installed as the

3DS GPU. It employs a framebuffer architecture. **There are no equivalents of the OBJ and BG concepts** previously employed on the Nintendo DS. Unlike rendering with the 3D engine for the Nintendo DS (which was closely connected with the VSync), on CTR rendering is possible until buffers are swapped. In other words, you can control the trade-off between the frame rate and rendering quality.

3D graphics features are based on OpenGL ES 1.1, but some of them correspond to OpenGL ES 2.0. Frequently used 3D graphics features are built into the hardware. Shader programs are supported. Programmable vertex shaders and non-programmable pixel shaders are also provided.

> **Note:** A simple comparison of the GPU reveals that its specifications do not match the Wii or Nintendo GameCube GPU in terms of vertex operations or fill rate. However, built-in hardware features allow richer imagery to be shown for some scenes than is possible with the Wii or Nintendo GameCube.

## 2.1. Built-In Hardware Features

Frequently used graphics features, including some that are not stipulated by the OpenGL ES standard, are built into the hardware. This makes it possible to express varied imagery in a small number of processing steps.

The following features have been provided.

- Silhouette generation
- Particle generation
- Procedural textures
- Fragment lighting
- Gas rendering
- Self-shadowing and soft shadowing
- Polygon subdivision

Shader programs and reserved uniform settings can use these features for graphics processing.

### 2.1.1. Silhouette Generation

*Silhouette generation* is a feature that uses the GPU to detect object edges and render only those edges. You can use this feature to outline a selected object for emphasis or to render soft shadows.

### 2.1.2. Particle Generation

*Particle generation* is a feature that renders a large number of random particles. You can use this feature to render explosions, snow, and other effects.

### 2.1.3. Procedural Textures

*Procedural textures* represent a feature that automatically generates textures using a combination of random noise and geometric patterns. This feature is a fast, low-cost way to generate textures with regular geometric patterns, in addition to textures that have some randomness in addition to a regular pattern, such as wood grain and marble.

### 2.1.4. Fragment Lighting

*Fragment lighting* is used on the 3DS system to light scenes on a per-fragment basis. This feature allows you to use bump mapping, apply environment maps, and run lighting calculations quickly.

### 2.1.5. Gas Rendering

*Gas rendering* is a feature used to render gaseous objects, while accounting for intersections and foreground/background relationships with polygon objects. This feature allows you to render surface boundaries with other objects more naturally than, for example, gaseous objects rendered by particles.

### 2.1.6. Self-Shadowing and Soft Shadows

Self-shadowing and soft shadows are supported by the 3DS system. These features allow an object to cast shadows on itself, and soft shadows to be rendered more naturally around the contours of an object.

### 2.1.7. Polygon Subdivision

Polygon subdivision is a feature that takes a group of vertices that are input according to fixed rules and automatically splits them into smooth polygons, using the GPU. This feature allows you to render objects with curved surfaces, given a small number of input vertices.

## 2.2. Rendering Pipeline

3D graphics for 3DS are processed in a series of steps known as the rendering pipeline.

Figure 2-1 shows a block view of the rendering pipeline and Table 2-1 shows what each process does. For more information, see the relevant pages for each process.

**Figure 2-1. Schematic View of the Rendering Pipeline**

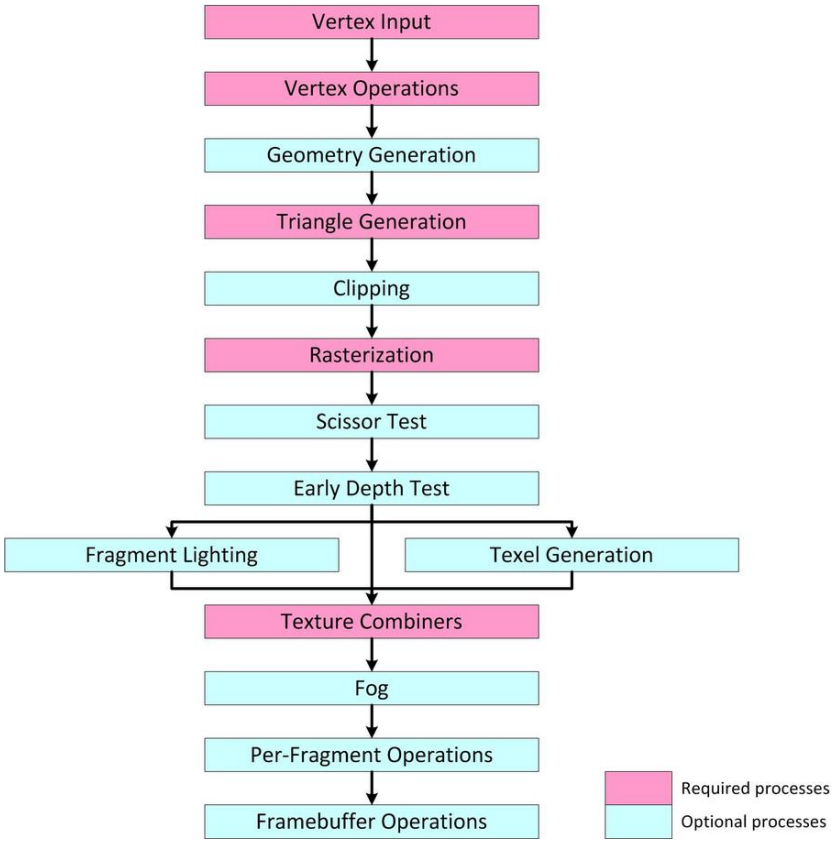



**Table 2-1. Actions Performed by Each Process**

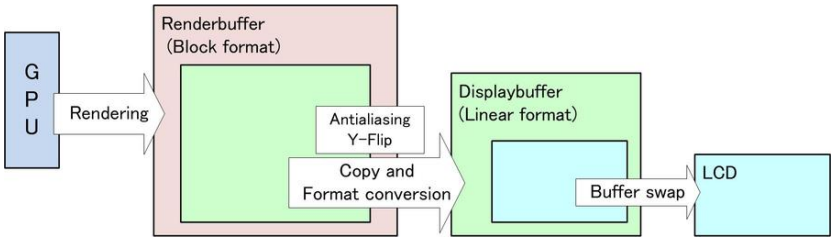
| Process | Description |
|---|---|
| Vertex input | Processes vertex data input from the application. |
| Vertex operations | Runs the vertex shaders. |
| Geometry generation | Runs the geometry shaders. |
| Triangle generation (Triangle setup) | Converts all primitives into triangles. |
| Clipping | Uses a clipping volume to clip primitives. |
| Rasterization | Converts primitives into fragments. |
| Scissor test | Eliminates unnecessary fragments via scissoring. |
| Early depth test | Runs depth tests before per-fragment operations. |
| Fragment lighting | Processes per-fragment lighting. |
| Texel generation | Determines texel colors from texture coordinates. |
| Texture combiners | Combines fragment light colors, texture colors, and so on. |
| Fog | Renders fog and gas. |
| Per-fragment operations | Runs tests, blending, logical operations, and other processing on fragments. |
| Framebuffer operations | Copies data from the framebuffer, runs pixel-read operations, and so on. |



# 3. LCD

On the 3DS system, the GX (graphics) library is closely involved with the images displayed on the LCD screens. Figure 3-1 shows the workflow starting from rendering and continuing until display on the LCD.

**Figure 3-1. Workflow From Rendering to Display on the LCD.**




The process is roughly split into the following steps.

1. Render.
2. Copy data from the render buffer to the display buffer and convert its format.
3. Swap buffers to update the regions displayed on the LCDs.

The rendering in step 1 is described in the documentation, starting from 4. Command Lists. This chapter describes the specifications for the output destination, the LCD, and the initialization needed when using the GX library. It also describes the allocation and transfer of buffers, synchronizing screen updates, and finalization, in that order.

> **Note:** For more information about stereoscopic display, see the *3DS Programming Manual: Advanced Graphics*.

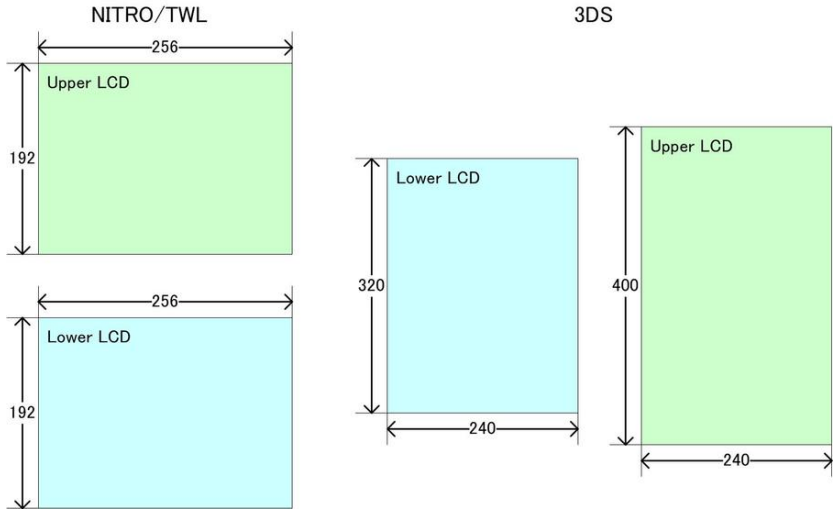
## 3.1. LCD Resolution, Placement, and Orientation

The LCDs on the 3DS system differ from the LCDs on the Nintendo DS (NITRO) and Nintendo DSi (TWL) systems, both in terms of resolution and arrangement. Table 3-1 shows the differences in resolution and Figure 3-2 shows the differences in placement and orientation.

**Table 3-1. LCD Resolution**

| LCD | NITRO/TWL | 3DS |
|---|---|---|
| Upper screen (width × height) | 256 × 192 | 240 × 400 |
| Lower screen (width × height) | 256 × 192 | 240 × 320 |

**Figure 3-2. LCD Placement and Orientation**




The standard LCD arrangement on the NITRO and TWL systems places the long edges of the LCDs (the screen width) horizontally, with the lower screen in front. The standard LCD arrangement on the 3DS system places the short edges of the LCDs horizontally, with the lower screen to the left of the upper screen (as when the front of the controller is rotated 90 degrees to the right).

# 3.2. Required Initialization Process

A framebuffer architecture has been adopted for 3DS. Because the images displayed on the LCDs are based on the framebuffer content, the GX library must be initialized first.

## 3.2.1. Initializing the GX Library

Call the `nngxInitialize()` function to initialize the GX library.

**Code 3-1. Initialization Function for the GX Library**

```
GLboolean nngxInitialize(
        GLvoid* (*allocator)(GLenum, GLenum, GLuint, GLsizei),
        void (deallocator)(GLenum, GLenum, GLuint, GLvoid));
GLboolean nngxGetIsInitialized();
```

An allocator and a deallocator are specified to the `nngxInitialize()` function, using the *`allocator`* and *`deallocator`* parameters, respectively. The allocator is called when a display buffer or other memory region is allocated, and the deallocator is called when the same memory region is freed. For more information about the allocator and deallocator, see 3.2.1.1. Allocator and 3.2.1.2. Deallocator.

A value of `GL_TRUE` is returned when initialization succeeds, and `GL_FALSE` is returned when initialization fails. A value of `GL_FALSE` is also returned if this function is called after the library has already been initialized but before the `nngxFinalize()` function has been called to shut down the library. Behavior is not guaranteed if any other `gl` or `nngx()` functions are called before using this function to initialize the library. The default render buffer, among others, is not allocated during initialization. These buffers must be allocated by the application after initialization.

The `nngxGetIsInitialized()` function returns `GL_TRUE` if the `nngxInitialize()` function

has initialized the GX library.

### 3.2.1.1. Allocator

The first allocator argument indicates the memory from which to allocate a region. The following values can be passed.

- `NN_GX_MEM_FCRAM`
  Allocate from main (device) memory.
- `NN_GX_MEM_VRAMA`
  Allocate from VRAM-A.
- `NN_GX_MEM_VRAMB`
  Allocate from VRAM-B.

Given a value of `NN_GX_MEM_FCRAM`, the allocator allocates from main memory, but the region to allocate must also be located in device memory. Device memory is a memory region that guarantees address consistency when it is accessed by peripheral devices. Applications are responsible for memory management. You can use the `nn::os::GetDeviceMemoryAddress()` and `nn::os::GetDeviceMemorySize()` functions to get the starting address and size of the memory region, allocated as device memory.

> **Note:** For more information about device memory, see the *3DS Programming Manual: System*.

If a value of `NN_GX_MEM_VRAMA` or `NN_GX_MEM_VRAMB` is specified for the allocator, use the `nn::gx::GetVramStartAddr`, `nn::gx::GetVramEndAddr`, and `nn::gx::GetVramSize()` functions to get the starting address, ending address, and size of VRAM, respectively. As with main memory, the application is responsible for memory management.

The second allocator argument is passed a value indicating the usage (buffer type) of the memory to allocate. Because addresses are aligned differently for each type, implement your application's allocator to comply with the following rules.

**Table 3-2. Differences in Alignment Between Buffer Types**

| Argument Value | Buffer Type | Alignment |
|---|---|---|
| `NN_GX_MEM_TEXTURE` | Texture (2D, environment map) | 128 bytes for all formats. |
| `NN_GX_MEM_VERTEXBUFFER` | Vertex buffers. | Depends on the vertex attribute.<br>4 bytes (`GLfloat` type)<br>2 bytes (`GLshort` type, `GLushort` type)<br>1 byte (`GLbyte` type, `GLubyte` type) |
| `NN_GX_MEM_RENDERBUFFER` | Color buffer. | 64 bytes |
| | Depth buffer (stencil buffer) | 32 bytes (16-bit depth)<br>96 bytes (24-bit depth)<br>64 bytes (24-bit depth + 8-bit stencil) |
| `NN_GX_MEM_DISPLAYBUFFER` | Display buffer | 16 bytes |
| `NN_GX_MEM_COMMANDBUFFER` | 3D command buffer | 16 bytes |

| | | |
|---|---|---|
| NN_GX_MEM_SYSTEM | System buffer | 4 bytes (when the memory size to allocate is a multiple of 4).<br>2 bytes (when the memory size to allocate is a multiple of 2, but not a multiple of 4).<br>1 byte (when none of the above). |

In addition to these rules, your implementation must comply with the following hardware specifications.

- All six faces of a cube map texture must fit within the same 32 MB boundaries.
- All six faces of a cube map texture must have addresses that share the same value for the most-significant 7 bits.
- The GL_TEXTURE_CUBE_MAP_POSITIVE_X face of a cube map texture must have a smaller address, or the same address, as every other face.
- Data must not be placed so that it spans VRAM-A and VRAM-B.

**Warning:** Do not allocate a display buffer in the last 1.5 MB of VRAM-A or VRAM-B.

If texture addresses are not correctly aligned, the GPU may hang, rendered results may be corrupted, or other problems may arise.

If NN_GX_MEM_VERTEXBUFFER, NN_GX_MEM_RENDERBUFFER, NN_GX_MEM_DISPLAYBUFFER, or NN_GX_MEM_COMMANDBUFFER was specified for the second argument, the third allocator argument is passed the name (ID) of that object.

The fourth allocator argument is passed the size of the region to allocate.

The application must account for address alignment and allocate the specified memory region based on these arguments, and then return the region's starting address. The application must return a value of 0 if it fails to allocate the region.

## 3.2.1.2. Deallocator

The first, second, and third deallocator arguments are passed the same values that were used when the memory region was allocated. The fourth argument is passed the starting address of the memory region. The application must use these arguments to release the memory region allocated by the allocator.

## 3.2.1.3. Getting an Allocator

You can get the allocator that was configured when the GX library was initialized by calling the nngxGetAllocator() function.

**Code 3-2. Getting an Allocator**

```
void nngxGetAllocator (
                GLvoid* (**allocator)(GLenum, GLenum, GLuint, GLsizei),
                void (*deallocator)(GLenum, GLenum, GLuint, GLvoid));
```

The *allocator* and *deallocator* parameters specify pointers that will be assigned pointers to the allocator and deallocator. No allocator or deallocator is obtained when specifying NULL for each of the arguments.

### 3.2.1.4. Getting an Initialization Register Setting Command

You can get an executable Initialization Register Setting Command when calling the nngxGetInitializationCommand() and nngxInitialize() functions.

> **Note:** This function was added as a measure related to rendering that takes place when returning to an application from the HOME Menu when generating a setting command directly in the register without using the graphics library supported by SDK. Consequently, it is not normally necessary to use it.

**Code 3-3. Getting an Initialization Register Setting Command**

```
GLsizei nngxGetInitializationCommand(GLsizei datasize, GLvoid* data);
```

For the *data* parameter, you can specify a pointer to the command that receives the register setting command. For the *datasize* parameter, you can specify the buffer byte size indicated by the *data* parameter.

This function returns the size in bytes of the Initialization Register Setting Command. Specifying 0 (NULL) for *data* gets the register setting command, and returns the buffer size to allocate. After you specify 0 for *data*, prepare that size of the buffer and get the command by making a call-out again.

If you are calling the nngxInitialize() function before initialization, 0 is returned without getting the command.

**Table 3-3. Error Generated by the nngxGetInitializationCommand Function**

| Error Code | Cause |
|---|---|
| GL_ERROR_80B4_DMP | The value specified for *datasize* is smaller than the obtained register command. |

## 3.2.2. Command-List Object Creation

You must create command list objects after initializing the GX library. Command list objects are introduced independently by the Nintendo 3DS, and are handled as execution units for 3D graphics processing. This section only explains how to create them. For more information about command lists and command list objects, see 4. Command Lists.

First use the nngxGenCmdlists() function to create one or more command list objects, individually bind them to the GPU with the nngxBindCmdlist() function, and then allocate individual memory regions with the nngxCmdlistStorage() function.

Command list objects comprise a 3D command buffer and command requests. Set *bufsize* and *requestcount* for the nngxCmdlistStorage() function to the size of the 3D command buffer and the number of command requests that can be queued. When you create multiple command list objects, you must call the nngxBindCmdlist() and nngxCmdlistStorage() function on each of

them.

The following code sample creates a single command list object that has a 3D command buffer size of 256 KB and can queue 128 command requests.

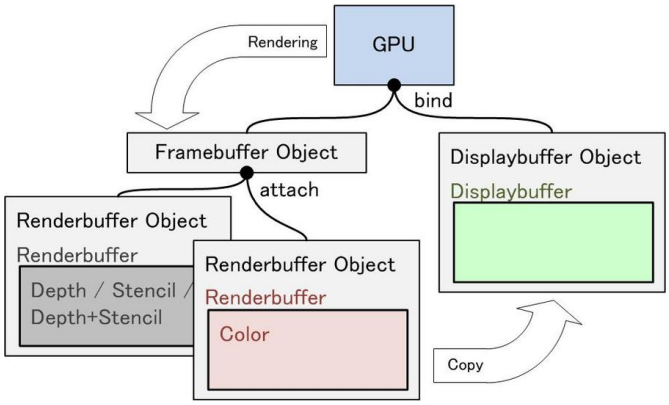
**Code 3-4. Creating a Command List Object**

```
GLuint commandList;

nngxGenCmdlists(1, &commandList);
nngxBindCmdlist(commandList);
nngxCmdlistStorage(256 * 1024, 128);
```

## 3.3. Allocating Buffers

Two of the buffers used graphics processing are involved in LCD output: the framebuffer, which is the GPU render target; and the display buffer, which is used to copy rendering results and display them on the LCDs. Processing to the framebuffer and the display buffer occur using the framebuffer object and the display buffer object, respectively. For the configuration, see Figure 3-3.

**Figure 3-3. Configuration of the Framebuffer and the Display Buffer**




### 3.3.1. Allocating Render Buffers

First, you must use the `glGenFramebuffers()` function to create a framebuffer object to specify as the render target. You can then bind the various render buffers (color, depth, or stencil) to the framebuffer object to specify those render buffers as the render target. The 3DS system has two screens (an upper and a lower screen), but if they have the same format and do not need to be rendered in parallel, we recommend that they share an object to conserve memory resources. In this case, specify the dimensions of the upper screen as the width and height of the buffer.

**Code 3-5. Creating a Framebuffer Object**

```
GLuint frameBufferObject;
glGenFramebuffers(1, &frameBufferObject);
```

Next, use the `glGenRenderbuffers()` function to create the render buffer objects. If the render target includes a depth buffer, stencil buffer, or both, in addition to a color buffer, you need two

render buffer objects for the framebuffer object.

**Code 3-6. Creating Render Buffer Objects**

```
GLuint renderBuffer[2];
glGenRenderbuffers(2, renderBuffer);
```

Use the `glBindRenderbuffer()` function to specify the render buffer object to bind to the framebuffer object, and then use the `glRenderbufferStorage()` function to allocate a render buffer.

**Code 3-7. Definition of the glRenderbufferStorage Function**

```
void glRenderbufferStorage(GLenum target, GLenum internalformat,
                           GLsizei width, GLsizei height);
```

Set *`width`* and *`height`* to the width and height of the render buffer. Neither the width nor the height can be greater than 1024 pixels.

> **Warning:** Block-shaped noise may be rendered on some pixels if the render buffer has 262,128 or more pixels (the product of its width and height). For more information, see 15.8. Block-Shaped Noise Is Rendered on Certain Pixels.

By setting *`target`* to the bitwise OR of `GL_RENDERBUFFER` and the following bitmasks, you can specify the memory from which to allocate the buffer. If no bitmask is specified, the buffer is allocated as if `NN_GX_MEM_VRAMA` was specified.

**Table 3-4. Specifiable Bitmasks for the target Parameter of glRenderbufferStorage**

| Bitmask | Buffer Is Allocated In |
|---|---|
| `NN_GX_MEM_VRAMA` | VRAM-A |
| `NN_GX_MEM_VRAMB` | VRAM-B |

Specify the buffer type (format) in the *`internalformat`* parameter. You can choose from the following formats on the 3DS system.

**Table 3-5. Specifiable Formats for the internalformat Parameter of glRenderbufferStorage**

| Format | Bits | Description of Format |
|---|---|---|
| `GL_DEPTH_COMPONENT16` | 16 | 16-bit depth. |
| `GL_DEPTH_COMPONENT24_OES` | 24 | 24-bit depth. |
| `GL_RGBA4` | 16 | The R, G, B, and alpha components are 4 bits each. |
| `GL_RGB5_A1` | 16 | The R, G, and B components are 5 bits each, and the alpha component is 1 bit. |
| `GL_RGB565` | 16 | 5-bit RB components and 6-bit G component. No alpha component. |
| `GL_RGBA8_OES` | 32 | The R, G, B, and alpha components are 8 bits each. |
| `GL_DEPTH24_STENCIL8_EXT` | 32 | 24-bit depth and 8-bit stencil value. |
| `GL_GAS_DMP` | 32 | Density values used during gas rendering (the rendered results cannot be copied to the display buffer). |

To specify an allocated render buffer as the render target, use the `glFramebufferRenderbuffer()` function to bind it to the framebuffer object that was specified by the `glBindFramebuffer()` function.

**Code 3-8. Definitions of glBindFramebuffer and glFramebufferRenderbuffer Functions**

```
void glBindFramebuffer(GLenum target, GLuint framebuffer);
void glFramebufferRenderbuffer(GLenum target, GLenum attachment,
        GLenum renderbuffertarget, GLuint renderbuffer);
```

Set *`target`* to `GL_FRAMEBUFFER` for both functions.

Set *`framebuffer`* to the framebuffer object and *`renderbuffer`* to the render buffer object. Set *`renderbuffertarget`* to `GL_RENDERBUFFER`.

The value to specify for *`attachment`* depends on the render buffer format.

**Table 3-6. Render Buffer Formats and the attachment Values to Use With Them**

| Format | Value of attachment |
|---|---|
| `GL_DEPTH_COMPONENT16` | `GL_DEPTH_ATTACHMENT` |
| `GL_DEPTH_COMPONENT24_OES` | |
| `GL_RGBA4` | `GL_COLOR_ATTACHMENT0` |
| `GL_RGB5_A1` | |
| `GL_RGB565` | |
| `GL_RGBA8_OES` | |
| `GL_GAS_DMP` | |
| `GL_DEPTH24_STENCIL8_EXT` | `GL_DEPTH_STENCIL_ATTACHMENT` |

You can call the `glCheckFramebufferStatus()` function to get the state of the render buffer object that is bound to the framebuffer object. Set the function's target parameter to `GL_FRAMEBUFFER`. A `GL_INVALID_ENUM` error is generated if you specify any other value.

If `GL_FRAMEBUFFER_COMPLETE` is returned, a proper render buffer object has been bound.

If `GL_FRAMEBUFFER_INCOMPLETE_MISSING_ATTACHMENT` is returned, neither the color buffer nor the depth (stencil) buffer has been attached.

If `GL_FRAMEBUFFER_INCOMPLETE_ATTACHMENT` is returned, either memory has not been allocated for the buffer that is bound, or the same render buffer object has been bound to both the color buffer and the depth buffer.

If `GL_FRAMEBUFFER_INCOMPLETE_DIMENSIONS` is returned, buffers with different sizes have been bound.

The following sample code allocates render buffers. Note that settings differ between the color and depth (stencil) buffers. The render buffer is shared by the upper and lower screens, so its width and height are specified using the dimensions of the upper screen.

**Code 3-9. Allocating Render Buffers (Color, Depth, Stencil)**

```
// FrameBuffer
glBindFramebuffer(GL_FRAMEBUFFER, frameBufferObject);
// Color
glBindRenderbuffer(GL_RENDERBUFFER, renderBuffer[0]);
glRenderbufferStorage(GL_RENDERBUFFER | NN_GX_MEM_VRAMA, GL_RGBA8_OES,
    nn::gx::DISPLAY0_WIDTH, nn::gx::DISPLAY0_HEIGHT);
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
    GL_RENDERBUFFER, renderBuffer[0]);
// Depth / Stencil
glBindRenderbuffer(GL_RENDERBUFFER, renderBuffer[1]);
```

```
glRenderbufferStorage(GL_RENDERBUFFER | NN_GX_MEM_VRAMB,
    GL_DEPTH24_STENCIL8_EXT, nn::gx::DISPLAY0_WIDTH, nn::gx::DISPLAY0_HEIGHT);
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_STENCIL_ATTACHMENT,
    GL_RENDERBUFFER, renderBuffer[1]);
```

## 3.3.2. Allocating Display Buffers

Before you allocate a display buffer, you must call the `nngxActiveDisplay()` function to specify whether to use it for the upper or lower screen.

**Code 3-10. Definition of the nngxActiveDisplay Function**

```
void nngxActiveDisplay(GLenum display);
```

Specify either the upper or lower screen with the value you pass to *`display`*. A `GL_ERROR_801F_DMP` error occurs when *`display`* is set to a value other than the ones listed in Table 3-7.

**Table 3-7. Values to Set for display**

| Value of display | Specified Screen |
|---|---|
| `NN_GX_DISPLAY0` | Upper screen (during stereoscopic display, this is the image for the left eye). |
| `NN_GX_DISPLAY0_EXT` | Upper screen (specifiable only during stereoscopic display, when this is the image for the right eye). |
| `NN_GX_DISPLAY1` | Lower screen. |

**Note:** For more information about stereoscopic display, see the *3DS Programming Manual: Advanced Graphics*.

Next, use the `nngxGenDisplaybuffers()` function to create the display buffer objects.

**Code 3-11. Definition of the nngxGenDisplaybuffers Function**

```
void nngxGenDisplaybuffers(GLsizei n, GLuint* buffers);
```

Set *`n`* to the number of display buffer objects to be created. Set *`buffers`* to point to an array in which to store the display buffer objects. If you are using more than one display buffer for a single screen through multi-buffering, create only as many objects as you need.

**Table 3-8. Errors Generated by the nngxGetInitializationCommand Function**

| Error Code | Cause |
|---|---|
| `GL_ERROR_801C_DMP` | A negative value was specified for *`n`*. |
| `GL_ERROR_801D_DMP` | The internal buffer failed to be allocated. |

Call the `nngxBindDisplaybuffer()` function on the generated display buffer object to set it as the target display buffer.

**Code 3-12. Definition of the nngxGenDisplaybuffers Function**

```
void nngxBindDisplaybuffer(GLuint buffer);
```

An object is created as long as an unused object name is specified for *buffer*. A GL_ERROR_8020_DMP error occurs when the internal buffer failed to be allocated.

Use the nngxDisplaybufferStorage() function to allocate a display buffer.

**Code 3-13. Definition of the nngxDisplaybufferStorage Function**

```
void nngxDisplaybufferStorage(GLenum format, GLsizei width, GLsizei height,
                              GLenum area);
```

For *format* specify one of the following buffer formats. You cannot specify a format that uses more bits per pixel than the format set when the color buffer was allocated.

**Table 3-9. Specifiable Formats for format of nngxDisplaybufferStorage**

| Format | Bits | Description of Format |
|---|---|---|
| GL_RGBA4 | 16 | The R, G, B, and alpha components are 4 bits each. |
| GL_RGB5_A1 | 16 | The R, G, and B components are 5 bits each, and the alpha component is 1 bit. |
| GL_RGB565 | 16 | 5-bit RB components and 6-bit G component. No alpha component. |
| GL_RGB8_OES | 24 | 8-bit RGB components. No alpha component. |

For *width* and *height*, specify the display buffer size. Both dimensions must be positive numbers that are multiples of the block size. The block size is either 8 or 32, depending on the render buffer's current block mode. For information about block mode settings, see the *3DS Programming Manual: Advanced Graphics*.

For *area*, specify the memory in which to allocate the buffer.

**Table 3-10. Specifiable Flags for area of nngxDisplaybufferStorage Function**

| Flag | Buffer Is Allocated In |
|---|---|
| NN_GX_MEM_FCRAM | Main (device) memory. |
| NN_GX_MEM_VRAMA | VRAM-A |
| NN_GX_MEM_VRAMB | VRAM-B |

To capture and save a screen image, the display buffer must be allocated in main memory where it can be accessed by the CPU.

If a display buffer object already has a display buffer allocated and you allocate a new display buffer for it, the old memory region is freed and a new memory region is allocated.

**Table 3-11. Errors Generated by the nngxGetInitializationCommand Function**

| Error Code | Cause |
|---|---|
| GL_ERROR_8021_DMP | The target is a display buffer whose object name is 0. |
| GL_ERROR_8022_DMP | An invalid value is specified for *width* or *height*. |
| GL_ERROR_8023_DMP | An invalid value is specified for *format*. |
| GL_ERROR_8024_DMP | An invalid value is specified for *area*. |
| GL_ERROR_8025_DMP | The memory region failed to be allocated. |

You can allocate multiple display buffers. The following sample code uses double-buffering.

**Code 3-14. Allocating Display Buffers**

```
GLuint display0Buffers[2];
GLuint display1Buffers[2];
// Displaybuffer for UpperLCD
nngxActiveDisplay(NN_GX_DISPLAY0);
nngxGenDisplaybuffers(2, display0Buffers);
nngxBindDisplaybuffer(display0Buffers[0]);
nngxDisplaybufferStorage(GL_RGB8_OES, nn::gx::DISPLAY0_WIDTH,
        nn::gx::DISPLAY0_HEIGHT, NN_GX_MEM_FCRAM);
nngxBindDisplaybuffer(display0Buffers[1]);
nngxDisplaybufferStorage(GL_RGB8_OES, nn::gx::DISPLAY0_WIDTH,
        nn::gx::DISPLAY0_HEIGHT, NN_GX_MEM_FCRAM);
nngxDisplayEnv(0, 0);
// Displaybuffer for LowerLCD
nngxActiveDisplay(NN_GX_DISPLAY1);
nngxGenDisplaybuffers(2, display1Buffers);
nngxBindDisplaybuffer(display1Buffers[0]);
nngxDisplaybufferStorage(GL_RGB8_OES, nn::gx::DISPLAY1_WIDTH,
        nn::gx::DISPLAY1_HEIGHT, NN_GX_MEM_FCRAM);
nngxBindDisplaybuffer(display1Buffers[1]);
nngxDisplaybufferStorage(GL_RGB8_OES, nn::gx::DISPLAY1_WIDTH,
        nn::gx::DISPLAY1_HEIGHT, NN_GX_MEM_FCRAM);
nngxDisplayEnv(0, 0);
```

### 3.3.2.1. Getting Parameters

You can use the `nngxGetDisplaybufferParameteri()` function to get information about the target display buffer.

**Code 3-15. Function for Obtaining Display Buffer Parameters**

```
void nngxGetDisplaybufferParameteri(GLenum pname, GLint* param);
```

You can set *pname* to one of the following values. Specifying any other value results in a `GL_ERROR_8033_DMP` error.

**Table 3-12. Display Buffer Parameters**

| pname | Parameter Information to Get |
|---|---|
| `NN_GX_DISPLAYBUFFER_ADDRESS` | Starting address of the display buffer. |
| `NN_GX_DISPLAYBUFFER_FORMAT` | The display buffer format. |
| `NN_GX_DISPLAYBUFFER_WIDTH` | The display buffer width. |
| `NN_GX_DISPLAYBUFFER_HEIGHT` | The display buffer height. |

## 3.4. Copying From the Color Buffer to the Display Buffer

When rendering is done, the content of the color buffer is in block format. This cannot be output to the LCDs, which can only display data in a linear format. When rendering has finished, the content of the color buffer is in block format. This data cannot be output to the LCDs, which can only display data in a linear format. You must call the `nngxTransferRenderImage()` function to convert the data into a format that can be output to the LCDs. This function also copies the data to the display buffer from the color buffer that is bound to the framebuffer object specified by `glBindFramebuffer`.

**Code 3-16. Function for Copying Data From the Color Buffer to the Display Buffer**

```
void nngxTransferRenderImage(GLuint buffer, GLenum mode, GLboolean yflip,
                             GLint colorx, GLint colory);
```

The *buffer* parameter specifies the display buffer object into which to copy the data. If the 3D command buffer has accumulated unsplit commands, the function adds a split command and a transfer command request to the command requests. There is no guarantee that any added transfer commands have completed running after this function has executed. Wait until the command list has finished executing before performing other tasks such as deleting color buffers or changing buffer contents.

The *mode* parameter specifies the degree of anti-aliasing to apply when the rendered results are copied to the display buffer.

**Table 3-13. Anti-Aliasing Specifications**

| mode | Anti-Aliasing Specification | Width | Height |
|---|---|---|---|
| NN_GX_ANTIALIASE_NOT_USED | Off (none). | Equal | Equal |
| NN_GX_ANTIALIASE_2x1 | 2×1 anti-aliasing. | 2 times | Equal |
| NN_GX_ANTIALIASE_2x2 | 2×2 anti-aliasing. | 2 times | 2 times |

The value of *yflip* specifies whether to apply a y-flip (a vertical flip) when the rendered results are copied to the display buffer. It is applied if the argument is set to GL_TRUE. Any value other than 0 is treated as GL_TRUE.

The values of *colorx* and *colory* specify the offsets to use when copying the data from the color buffer. (The function assumes that the origin is at the lower-left corner and that the positive axes are pointing up and right.) You must specify offsets that are positive multiples of the block size, which is 8 in block-8 mode and 32 in block-32 mode. (For more information about block mode settings, see the 3DS Programming Manual: Advanced Graphics.)

Starting from the offset position in the color buffer, this function copies a region with the same width and height as the display buffer to the display buffer. Subtract the offset values from the color buffer's width and height to find the dimensions of the region that can be copied from the color buffer.

The height and width of the region to copy, as measured in pixels, must be at least as big as the minimum allowed. The minimum height and width when copying from a color buffer is 128. The minimum height and width when copying to a display buffer depends on the anti-alias setting. If anti-aliasing is disabled, the minimum for both height and width is 128. If 2x1 anti-aliasing is enabled, the height minimum is 128 and the width minimum is 64. If 2x2 anti-aliasing is enabled, the minimum for both height and width is 64.

**Table 3-14. Errors Generated by the nngxTransferRenderImage Function**

| Error Code | Cause |
|---|---|
| GL_ERROR_8027_DMP | Called when the bound command list's object name is 0. |
| GL_ERROR_8028_DMP | The command request has already reached the maximum number of accumulated command requests allowable. |
| GL_ERROR_8029_DMP | A valid display buffer has not been bound. |
| GL_ERROR_802A_DMP | A valid color buffer has not been bound. |
| GL_ERROR_802B_DMP | An invalid value was specified for *mode*. |
| GL_ERROR_802C_DMP | The display buffer is larger than the region that can be copied. (When anti-aliasing, apply the width and height ratios from Table 3-13.) |
| GL_ERROR_802D_DMP | Invalid values were specified for *colorx* and *colory*. |
| GL_ERROR_802E_DMP | The display buffer uses more bits per pixel than the color buffer. |

| | |
|---|---|
| GL_ERROR_802F_DMP | The 3D command buffer is full because of commands added by this function. |
| GL_ERROR_8059_DMP | The width and height of the color buffer and display buffer are not multiples of 32 in block-32 mode. |
| GL_ERROR_805A_DMP | The width of the color buffer and display buffer are not multiples of 16 when the display buffer's pixel size is 24 bits and the block mode is block 8. |
| GL_ERROR_80B5_DMP | The specified height or width for copying from a color buffer was smaller than the minimum. |
| GL_ERROR_80B6_DMP | The specified height or width for copying to a display buffer was smaller than the minimum. |

## 3.5. Updating the LCD Render Regions by Swapping Buffers

After data has finished being copied to the display buffer, the buffer-swap function displays the rendered results to one or both LCDs.

Use the nngxActiveDisplay() function to specify a display and the nngxBindDisplaybuffer() function to bind the display buffer from which to display. You can then call the nngxDisplayEnv() function to specify the offsets to use when data is output from the display buffer to the LCDs. The function assumes that the origin is at the lower-left corner and the positive axes are pointing up and right, and that they are always positive values. Set (0, 0) for the offsets, if the display buffer is the same size as the LCD. Specifying a negative value for the offset results in a GL_ERROR_8026_DMP error.

**Code 3-17. Functions for Specifying the Display and the Display Buffer**

```
void nngxActiveDisplay(GLenum display);
void nngxBindDisplaybuffer(GLuint buffer);
void nngxDisplayEnv(GLint displayx, GLint displayy);
```

Next, use the buffer-swap function to switch the buffer that is output to the LCD.

**Code 3-18. Buffer-Swap Function**

```
void nngxSwapBuffers(GLenum display);
```

The nngxSwapBuffers() function swaps the buffers during the next VSync. This function can be called at any time, but if it is called more than once before a VSync, only the last call is valid.

Use *display* to specify which display is affected when the buffers are swapped. Specify NN_GX_DISPLAY0 to target only the upper screen, NN_GX_DISPLAY1 to target only the lower screen, orNN_GX_DISPLAY_BOTH to target both screens.

This function configures the GPU with the address of the display buffer to display, switching the image that is displayed on the LCDs. The display buffer address that is ultimately set in the GPU is calculated from the starting address of the buffer allocated by the nngxDisplaybufferStorage() function. The calculation takes into account several variables, including the display buffer resolution, the pixel size or number of bits per pixel (bpp), the LCD resolution, and the offsets configured by the nngxDisplayEnv() function.

The following equation calculates the address to configure.

BufferAddress + PixelSize × (DisplayBufferWidth × (DisplayBufferHeight - LcdHeight - DisplayY)

+ DisplayX)

**Table 3-15. Errors Generated by the nngxTransferRenderImage Function**

| Error Code | Cause |
|---|---|
| GL_ERROR_8030_DMP | An invalid value was specified for *display*. |
| GL_ERROR_8031_DMP | A valid display buffer has not been bound. |
| GL_ERROR_8032_DMP | The display region that considered the offset will fall outside of the display buffer. |
| GL_ERROR_8053_DMP | The display buffer address set in the GPU is not 16-byte aligned. |
| GL_ERROR_9000_DMP | The display buffer bound to the upper screen for the right eye during stereoscopic display (NN_GX_DISPLAY0_EXT) is 0, or that region has not been allocated. |
| GL_ERROR_9001_DMP | The nngxDisplayEnv() function specifies a display region that is outside the display buffer. |
| GL_ERROR_9002_DMP | The display buffers bound to the two upper screens (NN_GX_DISPLAY0 and NN_GX_DISPLAY0_EXT) have different resolutions, formats, or memory regions. |

In the system's initial state, a black screen is forced to be displayed on the LCDs. If a valid display buffer for LCD output has been prepared in the buffer-swap function, call the nngxStartLcdDisplay() function at the same time as a VSync to start LCD output. You only need to call this function the first time.

**Code 3-19. Function for Starting LCD Output**

```
void nngxStartLcdDisplay( void );
```

## 3.5.1. Swapping Buffers for a Specified Address

By calling the nngxSwapBuffersByAddress() function, you can specify the address of a buffer to display (on the LCDs) without using a display buffer object.

**Code 3-20. Swapping Buffers for a Specified Address**

```
void nngxSwapBuffersByAddress(GLenum display,
                              const GLvoid* addr, const GLvoid* addrB,
                              GLsizei width, GLenum format);
```

Buffers are swapped during the first VSync after this function (like the nngxSwapBuffers() function) is called. If this function is called more than once on the same display before a VSync occurs, only the last function call is valid.

Use *display* to specify which display is affected when the buffers are swapped. Specify NN_GX_DISPLAY0 for the upper screen or NN_GX_DISPLAY1 for the lower screen.

Use *addr* to specify the address of the buffer to display. If stereoscopic display is enabled and you have specified the upper screen with display, this argument provides the address of images to display for the left eye. You must specify an address that is 16-byte aligned.

Use *addrB* to specify the address of images to display for the right eye when stereoscopic display is enabled. This argument is only valid when you have specified the upper screen with display. It is ignored if stereoscopic display is disabled or if you have specified NN_GX_DISPLAY1 in *display*. You must specify an address that is 16-byte aligned.

The display position specified by the nngxDisplayEnv() function is ignored when the buffer specified by this function is displayed. Consequently, consider offsets and other factors in the addresses that you specify in *addr* and *addrB*. For more information about how addresses are calculated, see 3.5. Updating the LCD Render Regions by Swapping Buffers.

Use *width* to specify the width (in pixels) of the buffer to display. Note that *width* specifies the width of the buffer rather than the width of the LCD. Both the upper and lower screens have a width of 240 pixels, but if you want to partially display a buffer that is wider than that, specify the width of the entire buffer (in pixels), including the parts that will not be displayed. *width* must be a multiple of 8 and cannot be less than 240.

Use *format* to specify the buffer format. You can specify the same formats that can be specified when allocating a display buffer (see Table 3-9).

**Table 3-16. Errors Generated by the nngxSwapBuffersbyAddress Function**

| Error Code | Cause |
|---|---|
| GL_ERROR_8087_DMP | An invalid value was specified for *display*. |
| GL_ERROR_8088_DMP | The address specified in *addr* is not 16-byte aligned. |
| GL_ERROR_8089_DMP | The address specified in *addrB* is not 16-byte aligned. |
| GL_ERROR_808A_DMP | An invalid value was specified for *width*. |
| GL_ERROR_808B_DMP | A value other than a specifiable format was specified for *format*. |

## 3.5.2. Details of Swapping Buffers

The nngxSwapBuffers() and nngxSwapBuffersByAddress() functions do not show the content of the display buffer after they swap buffers. These functions simply schedule the display buffer address (used when the screen is rendered after a VSync) to be changed during a VBlank.

The GPU displays the content of the display buffer to the LCD. Because the GPU reads a single line of data from the display buffer for each scan line, memory is accessed frequently outside of a VBlank. This causes tearing to occur if the content of the display buffer is overwritten outside of a VBlank.

The GPU accesses memory at the highest priority when it transfers (displays) the content of the display buffer to the LCD. By placing the display buffer in VRAM that can only be accessed by the GPU, you can avoid problems caused by memory access conflicts. If, however, you place the display buffer in main memory (device memory), there may be memory access conflicts with the CPU and other devices.

You can use the nngxSetMemAccessPrioMode() function to adjust the priority at which the GPU, CPU, and other devices access main memory.

**Code 3-21. Function for Adjusting the Priority of Main Memory Access**

```
void nngxSetMemAccessPrioMode(nngxMemAccessPrioMode mode);
```

**Table 3-17. Differences in Main Memory Access Priorities**

| mode | GPU | CPU | Other Devices |
|---|---|---|---|
| NN_GX_MEM_ACCESS_PRIO_MODE_0 | Uniform priority. | | |
| NN_GX_MEM_ACCESS_PRIO_MODE_1 | | High priority. | |
| NN_GX_MEM_ACCESS_PRIO_MODE_2 | | Extremely high priority. | |
| NN_GX_MEM_ACCESS_PRIO_MODE_3 | High priority. | High priority. | |

| | | | |
|---|---|---|---|
| NN_GX_MEM_ACCESS_PRIO_MODE_4 | High priority. | | |

NN_GX_MEM_ACCESS_PRIO_MODE_1 is the default setting.

By raising the priority of memory accesses from the CPU, you can reduce the effect of the GPU and other devices on the time taken by processes that involve accessing main memory from the CPU.

**Warning:** If you place the display buffer in main memory and specify NN_GX_MEM_ACCESS_PRIO_MODE_2, a large number of memory accesses from the CPU may cause noise resembling vertical lines to appear on the screen because of insufficient bandwidth for transferring images to be displayed on the LCD. To avoid this, either place the display buffer in VRAM or specify another mode.

**Note:** If there is insufficient bandwidth for displaying images on the LCD, the nngxGetCmdlistParameteri() function returns a bit array with a value of 1 stored in both bit 17 and bit 18 when NN_GX_CMDLIST_HW_STATE is specified for *param*. For information about the nngxGetCmdlistParameteri() function, see 4.1.10. Getting Command List Parameters.

## 3.6. Screen Update Synchronization

You can use the following functions for processing that needs to line up with vertical LCD synchronization (VSync).

**Code 3-22. VSync Functions**

```
GLint nngxCheckVSync(GLenum display);
void nngxWaitVSync(GLenum display);
void nngxSetVSyncCallback(GLenum display, void (*func)(GLenum));
```

The nngxCheckVSync() function returns the value of the VSync counter updated within the library. By checking for changes in this value you can asynchronously check for VSync updates. Because the return value is an update counter used internally by the library, it wraps around to 0 after it exceeds the implementation's maximum value. (This may change in future implementations.)

The nngxWaitVSync() function waits for a VSync update on the specified LCD. Control does not return until the VSync update.

The nngxSetVSyncCallback() function registers the callback function that is invoked when VSync is updated. If this function is called with *func* set to 0 (NULL), it unregisters the callback function. The registered callback function is called from a different thread than the main thread, so mutual exclusion is required when referencing any data shared with the main thread. However, mutual exclusion is not required for data shared with any interrupt handlers registered using the nngxSetCmdlistCallback() function even if they are for the same graphics processing.

**Warning:** You can call the nngx() functions from within a VSync callback, but note that command request completion interrupts are forced to wait until the callback function completes. Consequently, minimize any calls within callback functions to functions that issue command requests.

Setting *display* to NN_GX_DISPLAY0, NN_GX_DISPLAY1, or NN_GX_DISPLAY_BOTH in any of these functions causes it to operate on the upper screen, lower screen, or both, respectively. Passing any other value causes a GL_ERROR_8019_DMP error in calls to nngxCheckVSync, a GL_ERROR_801A_DMP error in calls to nngxWaitVSync, and a GL_ERROR_801B_DMP error in calls to nngxSetVSyncCallback.

The system tries to prevent extreme time delay between screens, but there are approximately 100 microseconds between VSync updates for the upper and lower screens. Avoid creating code that strongly depends on this delay because heavy processing in the VSync callback for the upper screen or the start of an extremely high-priority thread may force the VSync callback for the lower screen to wait.

The LCD screen VSync interval is 59.831 Hz for both the upper and lower screens. This does not change when stereoscopic display (the parallax barrier) is enabled or disabled.

## 3.7. Finalizing

Call the nngxFinalize() function when you stop using the GX library (such as, when the application shuts down). This releases all remaining unreleased objects.

**Code 3-23. Definition of the nngxFinalize Function**

```
void nngxFinalize(void);
```

To destroy the framebuffer objects, render buffers, and display buffers allocated for displaying graphics on the LCDs, call glDeleteFramebuffers, glDeleteRenderbuffers, and nngxDeleteDisplaybuffers, respectively.

**Code 3-24. Functions for Destroying Framebuffer Objects, Render Buffers, and Display Buffers**

```
void glDeleteFramebuffers(GLsizei n, const GLuint* framebuffers);
void glDeleteRenderbuffers(GLsizei n, const GLuint* renderbuffers);
void nngxDeleteDisplaybuffers(GLsizei n, GLuint* buffers);
```

In each function, *n* specifies the number of object arrays passed to the second parameter. A GL_ERROR_801E_DMP error occurs when n is set to a negative value in the *nngxDeleteDisplaybuffers()*function.

If any of the specified framebuffer objects or render buffers is in use, it is unaffected. All other objects are deleted. Display buffers are swapped for display buffers with an object name of 0 and the objects in use are destroyed.

## 3.8. Specifying Display Portions

Figure 3-4 indicates which display portions are set to be specified for the transfer from the color buffer to the display buffer, and then to display on the LCD from the display buffer. The transfer from the color buffer to the display buffer assumes that anti-aliasing is disabled.

**Figure 3-4. Specifying Display Portions**

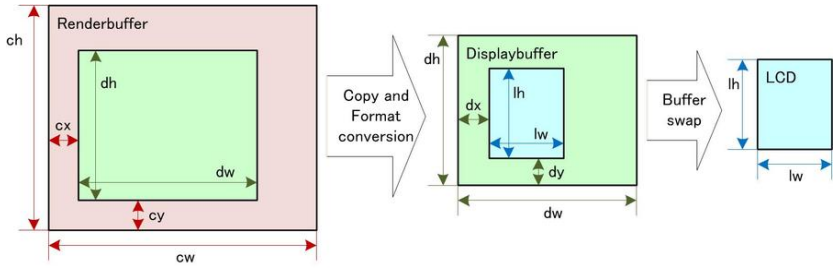



**Table 3-18. Specifying Display Portions**

| Variable Name | Description |
|---|---|
| cw, ch | Width and height of the color buffer<br>Values specified by *width* and *height* in the glRenderbufferStorage() function. (See Code 3-7.) |
| cx, cy | Offset for when copying from the color buffer to the display buffer.<br>Values specified by *colorx* and *colory* in the nngxTransferRenderImage() function. (See Code 3-16.) |
| dw, dh | Width and height of the display buffer<br>Values specified by *width* and *height* in the nngxDisplaybufferStorage() function. (See Code 3-13.) |
| dx, dy | Offset for when producing output from the display buffer to the LCD.<br>Values specified by *displayx* and *sy* in the nngxDisplayEnv() function. (See Code 3-17.) |
| lw, lh | The height and width of the output destination LCD. The height and width differ for the upper screen and lower screen.<br>See 3.1. LCD Resolution, Placement, and Orientation. |

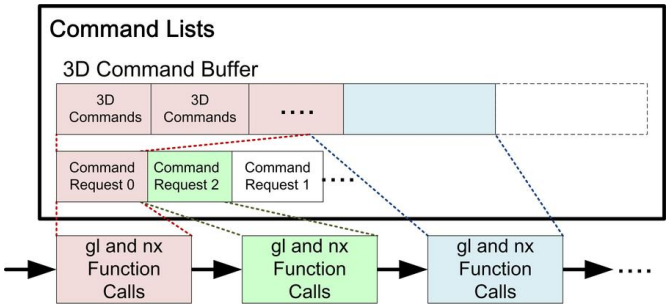
CONFIDENTIAL

# 4. Command Lists

Command lists are new to the Nintendo 3DS system. The gl and nngx functions called using 3D graphics processing can be recorded as commands and then executed all at the same time. Command list processing occurs using command list objects. The Nintendo 3DS handles command lists by the 3D graphics rendering execution unit.

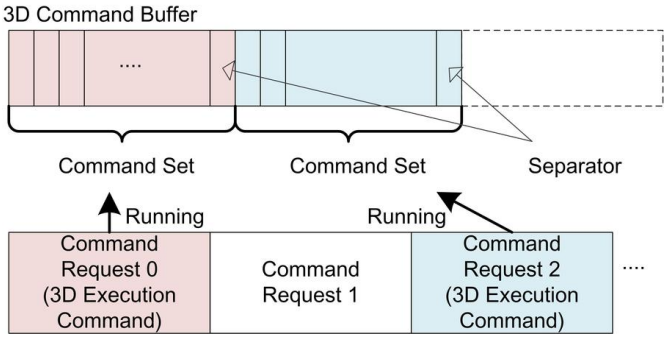
Command lists include commands that write to registers using direct GPU execution (3D commands), and command requests for communicating instructions from the CPU to the GPU. 3D commands accumulate in the 3D command buffer as gl() and nngx() functions carry out rendering work and other tasks. Command requests are queued by the specific gl() and nngx() functions that called them. For information about the types of command requests, see 4.2. Command Request Types.

#### 4-1. Command Lists




When a 3D execution command queued in the command request is processed, the GPU loads the 3D command from the command buffer and executes it. Multiple 3D commands are handled and run together as a single command set. Each command set ends with a split command, so the GPU can track where to end the loading of the 3D buffer.

**Figure 4-2. Command Sets**




**Note:** The function issuing command requests can only be called in the application core (Core 0).

# 4.1. How to Use

The Nintendo 3DS GPU renders 3D graphics by running commands in units of command lists. Applications create command list objects into which `gl()` functions and other functions accumulate 3D commands, which are then executed as one batch when the GPU runs the command list.

## 4.1.1. Creating Objects

First, use the `nngxGenCmdlists()` function to create the command list objects.

**Code 4-1. Function for Creating Command Lists**

```
void nngxGenCmdlists(GLsizei n, GLuint* cmdlists);
```

This code creates *n* command list objects and stores their object names in *`cmdlists`*.

Command lists have their own namespace. The command list with an object name of `0` is reserved by the system.

**Table 4-1. Errors Generated by the nngxGenCmdlists Function**

| Error Code | Cause |
|---|---|
| GL_ERROR_8000_DMP | A negative value was specified for *n*. |
| GL_ERROR_8001_DMP | The internal buffer failed to be allocated. |

## 4.1.2. Binding Command Lists

Next, use the nngxBindCmdlist() function to bind a generated command list object to the GPU. 3D commands are accumulated in the bound command list's 3D command buffer.

**Code 4-2. Function for Binding Command Lists**

```
void nngxBindCmdlist(GLuint cmdlist);
```

If *cmdlist* is set to an unused object name, that object is created.

**Table 4-2. Errors Generated by the nngxBindCmdlist() Function**

| Error Code | Cause |
|---|---|
| GL_ERROR_8004_DMP | The internal buffer failed to be allocated. |
| GL_ERROR_8005_DMP | Called while command caches and command lists were in a state of being saved. (For more information, see the CTR Programming Manual: Advanced Graphics.) |

## 4.1.3. Allocating Memory Regions

Use the nngxCmdlistStorage() function to allocate a memory region for the bound command list.

**Code 4-3. Allocating a Memory Region for a Command List**

```
void nngxCmdlistStorage(GLsizei bufsize, GLsizei requestcount);
```

Set *bufsize* to the size of the 3D command buffer and *requestcount* to the number of command requests that can be queued.

You must call the nngxBindCmdlist() and nngxCmdlistStorage() functions on each command list object that you create. Calls to these functions are ignored when a command list with an object name of 0 is bound. If you call this function again on an object that already has an allocated region, that region is freed and reallocated.

A GL_ERROR_COMMANDBUFFER_FULL_DMP error is generated by the relevant function if 3D commands have accumulated past this allocated 3D command buffer size or if the 3D command buffer has not been set. A GL_ERROR_COMMANDREQUEST_FULL_DMP error is generated by the relevant function if command requests have been queued past this maximum queue size or if the buffer has not been set.

**Table 4-3. Common Errors When Allocating a Memory Region for a Command List**

| Error Code | Cause |
|---|---|

| | |
|---|---|
| GL_ERROR_8006_DMP | Failed to allocate a memory region. |
| GL_ERROR_8007_DMP | Called on a command list that is being executed. |
| GL_ERROR_8008_DMP | A negative value was specified as an argument. |

## 4.1.4. Running Command List Objects

Call the nngxRunCmdlist() function to start executing command requests that have been queued in the bound command list.

**Code 4-4. Function for Executing Command Lists**

```
void nngxRunCmdlist(void);
void nngxRunCmdlistByID(GLuint cmdlist);
```

Execution is ignored if the bound command list has an object name of 0. Likewise, attempts to bind a different command list and run this function are ignored while command requests are executing.

After command requests have started executing, you can accumulate more commands in that same list or you can bind another command list and accumulate commands there. However, commands must be executed in the same order in which they were accumulated.

Accumulation of commands in an executing command list must occur in the application core. Undefined actions can result when accumulation occurs outside the application core.

The nngxRunCmdlistByID() function runs the command list specified by *cmdlist* rather than the currently bound command list. Besides running the specified command list, it works the same as the nngxRunCmdlist() function.

**Table 4-4. Common Errors When Running Command Lists**

| Error Code | Cause |
|---|---|
| GL_ERROR_8009_DMP | Called on a command list for which a memory region has not been allocated (nngxRunCmdlist()). |
| GL_ERROR_80B1_DMP | Called on a command list for which a memory region has not been allocated (nngxRuncmdlistByID()). |

### 4.1.4.1. Getting the State of an Executing Command List

Determine whether a command list is running using the nngxGetIsRunning() function.

**Code 4-5. Function for Determining Command List Execution State**

```
GLboolean nngxGetIsRunning(void);
```

This function returns GL_TRUE if a command list is currently running, regardless of whether the command list is currently bound.

Similarly, you can determine whether a command list is currently running by passing the NN_GX_CMDLIST_IS_RUNNING value for *pname* for the nngxGetCmdlistParameteri() function. This method can be used to determine whether the currently bound command list is running. For information about the nngxGetCmdlistParameteri() function, see 4.1.10. Getting Command List Parameters.

## 4.1.5. Destroying Command List Objects

You can call the `nngxDeleteCmdlists()` function to free command list objects that are no longer necessary.

**Code 4-6. Function for Destroying Command Lists**

```
void nngxDeleteCmdlists(GLsizei n, const GLuint* cmdlists);
```

This destroys the command list objects specified by the *n* object names in *`cmdlists`*. A `GL_ERROR_8003_DMP` error occurs if any of the specified command list objects are currently being executed while all other specified command list objects are being freed.

**Table 4-5. Errors Generated by the nngxDeleteCmdlist() Function**

| Error Code | Cause |
|---|---|
| `GL_ERROR_8002_DMP` | A negative value was specified for *n*. |
| `GL_ERROR_8003_DMP` | A command list included in *`cmdlists`* is still being executed. |

## 4.1.6. Stopping a Command List

Call either of the following functions to stop a command list that is being executed.

**Code 4-7. Functions for Stopping Command Lists**

```
void nngxStopCmdlist(void);
void nngxReserveStopCmdlist(GLint id);
```

When the `nngxStopCmdlist()` function is called, it waits for any executing command request to complete and then stops the command list. You cannot stop a command request after it has started executing (or is waiting to commence execution).

The `nngxReserveStopCmdlist()` function stops the command list immediately after the *`id`*$^{th}$ accumulated command request finishes executing.

Call the `nngxRunCmdlist()` function to resume a stopped command list. Note, however, that this function will be ignored if it is called after the instruction to stop the command list but before the executing command requests finish executing.

**Table 4-6. Errors Generated by the nngxReserveStopCmdlist() Function**

| Error Code | Cause |
|---|---|
| `GL_ERROR_800A_DMP` | Called on a command list that is being executed. |
| `GL_ERROR_800B_DMP` | Zero, a negative number, or a value that exceeds the maximum number of command requests was specified for *`id`*. |

## 4.1.7. Splitting the 3D Command Buffer

Use the `nngxSplitDrawCmdlist()` function to add a buffer loading complete command to the 3D

command buffer and start queuing render command requests. If a command list accumulates 3D commands while it is executing, its 3D commands are run up to the point at which they are split by this function.

**Code 4-8. Function for Splitting the 3D Command Buffer**

```
void nngxSplitDrawCmdlist(void);
```

Render command requests are not queued until the buffer loading complete command is added. In addition to this function, other functions also queue render command requests. Because functions such as `glClear()` and `glTexImage2D()` must stop 3D command execution, they each add a buffer loading complete command, and then queue the render command requests.

**Table 4-7. Common Errors Generated by the nngxSplitDrawCmdlist() Function**

| Error Code | Cause |
|---|---|
| `GL_ERROR_800C_DMP` | Called when the bound command list's object name is `0`. |
| `GL_ERROR_800D_DMP` | The command request has already reached the maximum number of accumulated command requests allowable. |
| `GL_ERROR_800E_DMP` | The 3D command buffer is full because of commands added by this function. |

These errors may also be generated by other functions that call this one internally.

## 4.1.7.1. Flushing the Accumulated 3D Command Buffer

When the `nngxSplitDrawCmdlist()` function is called, the buffer loading complete command is added and queuing of render command requests is carried out, even if there are no 3D commands accumulated in the 3D command buffer. In other words, this function can potentially add unneeded commands. We recommend calling the `nngxFlush3DCommand()` function (which adds a command to split the 3D command buffer) and the `nngxFlush3DCommandNoCacheFlush()` function only when 3D commands have accumulated. If a cache flush occurs multiple times, call a later function that reflects the cache content all at the same time by using the `nngxUpdateBufferLight()` function. This can reduce CPU overhead.

**Code 4-9. 3D Function for Flushing the Command Buffer**

```
void nngxFlush3DCommand(void);
void nngxFlush3DCommandNoCacheFlush(void);
```

These functions will not add a buffer loading complete command and render command request if there are no 3D commands accumulated in the 3D command buffer of the bound command list after the buffer has been split for the last time. This function only adds a buffer loading complete command and render command request when 3D commands have been accumulated in the buffer. If 3D commands are accumulating as they are executing, the 3D commands execute up to where the buffer was split by this function.

**Table 4-8. Errors Generated by the nngxFlush3DCommand() and nngxFlush3DCommandNoCacheFlush() Functions**

| Error Code | Cause |
|---|---|
| `GL_ERROR_8084_DMP`<br>`GL_ERROR_80AE_DMP` | Called when the bound command list's object name is 0. |

| | |
|---|---|
| GL_ERROR_8085_DMP<br>GL_ERROR_80AF_DMP | The command request has already reached the maximum number of accumulated command requests allowable. |
| GL_ERROR_8086_DMP<br>GL_ERROR_80B0_DMP | The 3D command buffer is full because of commands added by this function. |

## 4.1.7.2. Partially Flushing the Accumulated 3D Command Buffer

The `nngxFlush3DCommandPartially()` function is provided for executing a 3D command of a specified size. This function is an extension of the features provided with the `nngxFlush3DCommand()` function and can be called to correctly execute 3D commands, including the command buffer execution register kick command added to functions such as `nngxAdd3DCommand()`. For more information, see *3DS Programming Manual: Advanced Graphics*, in 8.8.23. Command Buffer Execution Registers (0x0238 – 0x023D).

**Code 4-10. Function for Partially Flushing the Command Buffer**

```
void nngxFlush3DCommandPartially(GLsizei buffersize);
```

Specify the size, in bytes, of the command buffer to be executed in *`buffersize`*. The number must be a multiple of 16.

The size must be correctly specified in *`buffersize`* from the address following the previous command flush to the first kick command (and including the kick command). If the wrong value is specified, the command is executed in an unintended order and the operation may not be able to complete properly.

Using the application, accurately perform cache flush on 3D commands accumulated from the previous command flush up to the point this function is called. Overall flushing of the cache must be performed after calling the function because commands that generate an interrupt are generated within this function. In addition, as a means of avoiding cache execution before the cache is flushed, this function cannot be called for command lists whose execution is in progress. Functions such as `glClear`, `nngxTransferRenderImage()`, and `glCopyTexImage2D()` execute flushes according to the same method as the `nngxFlush3DCommand()` function. Always perform a flush using this function before calling those functions.

When a kick command is added using the `nngxAddJumpCommand()` or `nngxAddSubroutineCommand()` function, the driver adjusts the size so that the command kicks the execution size up to the first kick command. It is unnecessary to call the `nngxFlush3DCommandPartially()` function when using those functions.

Note that when a partial flush is performed for a command buffer with a kick command added with the `nngxAddSubroutineCommand()` function, the execution size used is specified in *`buffersize`* instead of an execution size calculated by the driver.

**Table 4-9. Errors Generated by the nngxFlush3DCommandPartially() Function**

| Error Code | Cause |
|---|---|
| GL_ERROR_80A9_DMP | Called when the bound command list's object name is 0. |
| GL_ERROR_80AA_DMP | The command request has already reached the maximum number of accumulated command requests allowable. |
| GL_ERROR_80AB_DMP | The 3D command buffer is full because of commands added by this function. |
| GL_ERROR_80AC_DMP | A value less than `0` or that is not a multiple of 16 was specified in *`buffersize`*. |
| GL_ERROR_80AD_DMP | Called on a command list that is being executed. |

## 4.1.8. Clearing the Command List

The following function clears the command list and sets both the 3D command buffer and command request queue to the unused state (the state immediately after their memory regions are allocated).

**Code 4-11. Function for Clearing the Command List**

```
void nngxClearCmdlist(void);
```

**Table 4-10. Errors Generated by the nngxClearCmdlist Function**

| Error Code | Cause |
|---|---|
| GL_ERROR_800F_DMP | Called on a command list that is being executed. |

### 4.1.8.1. Clearing the Command List and Filling Its 3D Command Buffer

The following function clears the command list and initializes the 3D command buffer with the specified data. Both the 3D command buffer and command request queue enter the unused state.

**Code 4-12. Function for Clearing the Command List and Filling Its 3D Command Buffer**

```
void nngxClearFillCmdlist(GLuint data);
```

**Table 4-11 Errors Generated by the nngxClearFillCmdlist() Function**

| Error Code | Cause |
|---|---|
| GL_ERROR_8065_DMP | Called on a command list that is being executed. |

## 4.1.9. Setting Command List Parameters

You can call the `nngxSetCmdlistParameteri()` function to set command list parameters

**Code 4-13. Function for Setting Command List Parameters**

```
void nngxSetCmdlistParameteri(GLenum pname, GLint param);
```

**Table 4-12. Command List Parameters That Can Be Set**

| pname | Setting |
|---|---|
| | This parameter is set for individual command list objects and can have one of the following values.<br>• GL_TRUE: Update the additive blend results for rendering gas density information.<br>• GL_FALSE: Normal behavior (default)<br>If this parameter is GL_TRUE, when the `nngxSplitDrawCmdlist()` or `nngxFlush3DCommand()` function is called, additive blend results are updated for rendered gas density information after the accumulated render command requests have finished running. |

| | |
|---|---|
| NN_GX_CMDLIST_GAS_UPDATE | If this parameter is GL_FALSE, normal behavior is restored; that is, commands that update gas density information are only accumulated when necessary.<br>This setting takes effect depending on whether it is GL_TRUE when nngxSplitDrawCmdlist() or nngxFlush3DCommand() is called. A setting of GL_TRUE does not have any effect when render command requests are executed. This setting also does not affect render command requests accumulated by any function call other than nngxSplitDrawCmdlist() or nngxFlush3DCommand().<br>For more information about how additive blend results are updated for rendered gas density information, see the Gas Control Setting Registers section in the *3DS Programming Manual: Advanced Graphics*. |

**Table 4-13. Errors Generated by the nngxSetCmdlistParameteri() Function**

| Error Code | Cause |
|---|---|
| GL_ERROR_8015_DMP | Called on a command list that is being executed. |
| GL_ERROR_8016_DMP | An invalid value was specified for *pname* or *param*. |

## 4.1.10. Getting Command List Parameters

You can call the nngxGetCmdlistParameteri() function to get command list parameters.

**Code 4-14. Function for Getting Command List Parameters**

```
void nngxGetCmdlistParameteri(GLenum pname, GLint* param);
```

**Table 4-14. Getting Command List Parameters**

| pname | Parameter Obtained |
|---|---|
| NN_GX_CMDLIST_IS_RUNNING | The command list execution state.<br>GL_TRUE: The command list is currently being executed.<br>GL_FALSE: The command list is not currently being executed. |
| NN_GX_CMDLIST_USED_BUFSIZE | The number of bytes accumulated in the 3D command buffer. |
| NN_GX_CMDLIST_USED_REQCOUNT | The number of accumulated command requests. |
| NN_GX_CMDLIST_MAX_BUFSIZE | The maximum 3D command buffer size.<br>This value is specified as *bufsize* to the nngxCmdlistStorage() function. |
| NN_GX_CMDLIST_MAX_REQCOUNT | The maximum number of command requests.<br>This value was specified in the *requestcount* parameter to the nngxCmdlistStorage() function. |
| NN_GX_CMDLIST_TOP_BUFADDR | The starting address of the 3D command buffer. |
| NN_GX_CMDLIST_BINDING | The object name of the command list that is currently bound. |
| NN_GX_CMDLIST_RUN_BUFSIZE | The number of executed bytes in the 3D command buffer. |
| NN_GX_CMDLIST_RUN_REQCOUNT | The number of executed command requests. |
| NN_GX_CMDLIST_TOP_REQADDR | The starting address of the data region used for the command request queue. |
| | The type of command request that is currently executing or will be executed next.<br>The value returned in *param* depends on the state of the command list that is currently bound. If the command list is currently stopped, the return value is the type of command request that will be executed next. If the command list is |

| | |
|---|---|
| NN_GX_CMDLIST_NEXT_REQTYPE | executing, the return value is the type of command request that is currently executing. NULL is returned when all command requests have finished executing.<br>Command request types are defined by the following macros.<br>NN_GX_CMDLIST_REQTYPE_DMA: DMA transfer command request.<br>NN_GX_CMDLIST_REQTYPE_RUN3D: Render command request.<br>NN_GX_CMDLIST_REQTYPE_FILLMEM: Memory fill command request.<br>NN_GX_CMDLIST_REQTYPE_POSTTRANS: Post transfer command request.<br>NN_GX_CMDLIST_REQTYPE_COPYTEX: Copy texture command request. |
| NN_GX_CMDLIST_NEXT_REQINFO | The command buffer's address and byte size.<br>The command buffer's address is stored in the first element of *param* and its size (in bytes) is stored in the second element. You must pass *param* a pointer to an array of at least two GLint values.<br>If the bound command list is currently stopped, parameter information is returned for the command request that will be executed next. If the bound command list is executing, parameter information is returned for the command request that is currently executing. Nothing is returned if all command requests have finished executing.<br>The function only returns information when a render command request is the command request that is currently executing or that will be executed next. Nothing is returned for any other type of command. |
| NN_GX_CMDLIST_HW_STATE | 32 bits of data indicating the hardware state.<br>Each of the following bits is set to 1 to indicate that the command list is in the specified state.<br>bit 20: A post transfer command request is executing.<br>bit 19: A memory fill is in progress.<br>bit 18: An underrun error has occurred in the FIFO for the lower screen.<br>bit 17: An underrun error has occurred in the FIFO for the upper screen.<br>bit 16: The post vertex cache is busy.<br>bit 15: Bits [1:0] of register 0x0252 are 1.<br>bit 14: Vertex processor 3 is busy.<br>bit 13: Vertex processor 2 is busy.<br>bit 12: Vertex processor 1 is busy.<br>bit 11: Vertex processor 0 (which can also be used as the geometry shader processor) is busy.<br>bit 10: Bits [1:0] of register 0x0229 are nonzero.<br>bit 9: Input to the module that loads command buffers and vertex arrays is busy.<br>bit 8: Output from the module that loads command buffers and vertex arrays is busy.<br>bit 7: The early depth test module is busy.<br>bit 6: The per-fragment operation module is busy processing data from the module in the previous stage.<br>bit 5: The per-fragment operation module is busy accessing the framebuffer.<br>bit 4: The texture combiners are busy.<br>bit 3: Fragment lighting is busy.<br>bit 2: The texture units are busy.<br>bit 1: The rasterization module is busy.<br>bit 0: Triangle setup is busy. |
| NN_GX_CMDLIST_CURRENT_BUFADDR | Buffer address of the next 3D command stored in the currently bound command list. |

**Table 4-15. Common Errors When Setting Command List Parameters**

| Error Code | Cause |
|---|---|
| GL_ERROR_8017_DMP | An invalid value was specified for *pname* or *param*. |

| | |
|---|---|
| GL_ERROR_8018_DMP | NN_GX_CMDLIST_BINDING is not specified for *pname* when the bound command list's object name is 0. |

## 4.1.11. Command Completion Interrupts

You can cause interrupts to occur and call interrupt handlers, when the command requests in a command list finish. You can register an interrupt handler with the nngxSetCmdlistCallback() function.

**Code 4-15. Function for Registering an Interrupt Handler**

```
void nngxSetCmdlistCallback(void (*func)(GLint));
```

An interrupt handler is valid only for the bound command list. If this function is called with *func* set to 0 (NULL), the handler is unregistered.

The interrupt handler is called from a different thread than the main thread, so mutual exclusion is needed when referencing any data shared with the main thread. However, mutual exclusion is not needed for data shared with any callback functions for the same graphics processing registered using the nngxSetVSyncCallback() function.

**Table 4-16. Common Errors When Setting Command List Callbacks**

| Error Code | Cause |
|---|---|
| GL_ERROR_8010_DMP | Called on a command list that is being executed. |

Use the nngxEnableCmdlistCallback() function to specify a command request that normally triggers an interrupt when it ends. The nngxDisableCmdlistCallback() function can disable interrupts.

**Code 4-16. Interrupt Control Functions**

```
void nngxEnableCmdlistCallback(GLint id);
void nngxDisableCmdlistCallback(GLint id);
```

An interrupt occurs upon completion of the $id^{th}$ accumulated command request. You can call this function on a single command list several times with separate *id* values to cause multiple interrupts to occur. Note that *id* indicates a command request in the order that it was accumulated, not in the order that it was executed. You can call nngxGetCmdlistParameteri() with *pname* set to NN_GX_CMDLIST_USED_REQCOUNT to get a value to specify for *id*. If *id* is −1, an interrupt occurs when all command requests accumulated in the command list have finished.

The command list is still executing when an interrupt handler is called. This occurs for every interrupt except for the last to the command request accumulated in the command list. Consequently, the interrupt handler cannot, itself, call any functions that cannot be called while a command list is executing.

Even without registering an interrupt handler, you can determine when a command request has finished executing by calling nngxGetCmdlistParameteri() passing *pname* as NN_GX_CMDLIST_IS_RUNNING, and then waiting until you get a value of GL_FALSE.

**Table 4-17. Common Errors When Enabling and Disabling Command List Callbacks**

| Error Code | Cause |
|---|---|

| | |
|---|---|
| GL_ERROR_8012_DMP<br>GL_ERROR_8014_DMP | Zero, a negative number other than –1, or a value equal to the maximum number of command requests was specified for *id*. |

## 4.1.12. Waiting for Command Execution to Complete

You can call nngxWaitCmdlistDone to wait for all of the command requests accumulated in the command list to complete.

**Code 4-17. Function That Waits for Commands to Complete**

```
void nngxWaitCmdlistDone(void);
```

Render command requests are executed until the point at which they are split. To execute all of the accumulated render command requests, call nngxSplitDrawCmdlist() before this function.

This function does not return until command execution is complete. However, you can use the nngxSetTimeout() function to set a timeout period.

**Code 4-18. Function for Setting a Timeout When Waiting for Command Execution to Complete**

```
void nngxSetTimeout(GLint64EXT time, void (*callback)(void));
```

Set *time* to the number of ticks to wait before the nngxWaitCmdlistDone() function times out. Timeouts do not occur when a value of 0 is specified.

Set *callback* to the callback function to invoke when a timeout occurs. If this is NULL, a callback function is not invoked when the timeout occurs.

No timeouts occur by default because the initial values for *time* and *callback* are 0 and NULL respectively.

## 4.1.13. Adding a DMA Transfer Command Request

When the nngxAddVramDmaCommand() or nngxAddVramDmaCommandNoCacheFlush() function is called, a command request that runs a DMA transfer to VRAM is accumulated in the command list. The former flushes the source cache, but the latter does not. This function can only use DMA transfers from main memory to VRAM.

**Code 4-19. Function for Adding a DMA Transfer Command Request**

```
void nngxAddVramDmaCommand(
                        const GLvoid* srcaddr, GLvoid* dstaddr, GLsizei size);
void nngxAddVramDmaCommandNoCacheFlush(
                        const GLvoid* srcaddr, GLvoid* dstaddr, GLsizei size);
```

An amount of data specified by *size* is transferred from the address specified by *srcaddr* to the address specified by *dstaddr*.

When calling the nngxAddVramDmaCommand() function, a GL_ERROR_8062_DMP error indicates that this function was called when no valid command list was bound, while a GL_ERROR_8064_DMP error indicates that *size* is negative.

When calling the nngxAddVramDmaCommandNoCacheFlush() function, a GL_ERROR_8090_DMP error indicates that this function was called when no valid command list was bound, and a

GL_ERROR_8091_DMP error indicates that *size* is negative.

## 4.1.14. Adding an Anti-Aliasing Filter Transfer Command Request

When the nngxFilterBlockImage() function is called, a command request that transfers an image with an anti-aliasing filter applied is accumulated in the command list. (This is one kind of post-filter command request.) The image is transferred in block format, unconverted. The only supported antialiasing specification is 2×2.

**Code 4-20. Function for Adding an Anti-Aliasing Filter Transfer Command Request**

```
void nngxFilterBlockImage(const GLvoid* srcaddr, GLvoid* dstaddr,
                          GLsizei width, GLsizei height, GLenum format);
```

An image with a width, height, and format specified by *width*, *height*, and *format* respectively is transferred from the address specified by *srcaddr* to the address specified by *dstaddr*.

The *width* and *height* arguments are restricted as follows by the value specified for *format*.

**Table 4-18. Format Restrictions on the Width and Height of Images to Be Transferred**

| format | width | height |
|---|---|---|
| GL_RGBA8_OES<br>GL_RGB8_OES | A multiple of 64, greater than or equal to 64. | A multiple of 16, greater than or equal to 64. |
| GL_RGBA4<br>GL_RGB5_A1<br>GL_RGB565 | A multiple of 128, greater than or equal to 128. | A multiple of 16, greater than or equal to 128. |

If the transfer source and destination memory regions overlap, the function works properly when the *scraddr* and *dstaddr* values are the same, or when the *scraddr* value is bigger than the *dstaddr* value. The transfer results could be corrupted if the *scraddr* value is smaller than the *dstaddr* value.

When the value for *srcaddr* specifies an address in device memory, the transfer results could be incorrect if the destination memory cache has not been flushed.

**Table 4-19. Errors Generated by the nngxFilterBlockImage() Function**

| Error Code | Cause |
|---|---|
| GL_ERROR_8068_DMP | Called when a command list with an object name of 0 is bound or when there is no space in the command request queue. |
| GL_ERROR_8069_DMP | The address specified for *srcaddr* or *dstaddr* is not 8-byte aligned. |
| GL_ERROR_806A_DMP | A *width* or *height* value is specified that violates the restrictions. |
| GL_ERROR_806B_DMP | A *format* value is specified that is not listed in the restrictions. |

## 4.1.15. Adding an Image Transfer Command Request

When the nngxTransferLinearImage() function is called, a command request that transfers an image to a render buffer or texture is accumulated in the command list. (This is one kind of copy-texture command request.) If the current 3D command buffer has accumulated unsplit commands, a split command is added, and then the transfer command request is added.

Although images are converted from linear format to block format while they are transferred, this conversion only affects addressing. If this function is called on a render buffer, the block mode setting automatically determines whether a conversion to 8 block addressing or 32 block addressing is applied during the transfer. If this function is called on a texture, a conversion to 8 block addressing is applied. In either case, you must flip an image in the V direction and convert its byte order before you transfer it.

> **Note:** For information about block mode, see Block Mode Settings in the *3DS Programming Manual: Advanced Graphics*.

**Code 4-21. Function for Adding an Image Transfer Command Request**

```
void nngxTransferLinearImage(const GLvoid* srcaddr, GLuint dstid,
                             GLenum target);
```

For *srcaddr*, specify the starting address of the image to transfer. The image must have the same format, width, and height as the render buffer or texture to which it is transferred. However, the source pixel format must be 32-bit when the target pixel format is 24-bit because the hardware does not support transfers between 24-bit pixel formats. In this case, for each 4 bytes that are transferred, the first byte (the internal format's alpha component) is truncated.

The image is transferred to the render buffer or texture that has the object ID specified by *dstid* and the object type specified by *target*.

**Table 4-20. Values to Specify for target and dstid**

| When target is: | Set dstid to: |
|---|---|
| GL_RENDERBUFFER | The object ID of a render buffer.<br>If a value of 0 is specified, data is transferred to the color buffer that is attached to the current framebuffer. |
| GL_TEXTURE_2D | The object ID of a 2D texture. |
| GL_TEXTURE_CUBE_MAP_POSITIVE_X{,Y,Z}<br>GL_TEXTURE_CUBE_MAP_NEGATIVE_X{,Y,Z} | The object ID of a cube map texture. |

The width and height of the target render buffer must be multiples of 8, in block 8 mode or multiples of 32, in block 32 mode. Both the width and height must be at least 128.

**Table 4-21. Errors Generated by the nngxTransferRenderImage() Function**

| Error Code | Cause |
|---|---|
| GL_ERROR_805B_DMP | Called when the bound command list's object name is 0. |
| GL_ERROR_805C_DMP | The maximum number of command requests has already accumulated. |
| GL_ERROR_805D_DMP | The 3D command buffer is full because of commands added by this function. |
| GL_ERROR_805E_DMP | The render buffer or texture specified for *dstid* does not exist, or it does not have an allocated memory region. |
| GL_ERROR_805F_DMP | There is a violation of the width and height restrictions for the target render buffer. |
| GL_ERROR_8060_DMP | An invalid value was specified for *target*. |
| GL_ERROR_8067_DMP | The target render buffer or texture does not use 32-bit, 24-bit, or 16-bit pixel sizes. |

# 4.1.16. Adding a Block-to-Linear Image Conversion and Transfer Command Request

A command request for converting a block image to a linear image and transferring the result can be added to the command list by calling the `nngxAddB2LTransferCommand()` function. (This is one kind of post-filter command request.) Although the `nngxTransferRenderImage()` function provides the same functionality, the `nngxAddB2LTransferCommand()` function is more versatile. They also differ in that the latter function adds only a transfer request command and does not add a split command.

**Code 4-22. Function for Adding a Command Request for Converting From a Block Image to a Linear Image and Transferring**

```
void nngxAddB2LTransferCommand(
    const GLvoid* srcaddr, GLsizei srcwidth, GLsizei srcheight, GLenum
srcformat,
    GLvoid* dstaddr, GLsizei dstwidth, GLsizei dstheight, GLenum dstformat,
    GLenum aamode, GLboolean yflip, GLsizei blocksize);
```

The *srcaddr* parameter specifies the transfer source (block image) address. The *dstaddr* parameter specifies the transfer destination (linear image) address. Both *srcaddr* and *dstaddr* must be 16-byte aligned.

The *srcwidth*, *srcheight*, *dstwidth*, and *dstheight* parameters specify the transfer source image width and height and transfer destination width and height, in pixels. The height and width of the source image and destination image must be a multiple of the block size (8 or 32). Finally, if the pixel size of the destination image is 24 bits and the block size is 8, the width of the source image and width of the destination image must be a multiple of 16. If `0` is specified for *srcwidth*, *srcheight*, *dstwidth*, or *dstheight*, the command is not issued. The height and width of the destination image in pixels must be equal to, or less than, that of the source image.

The height and width of the source and destination images, as measured in pixels, must be at least as big as the minimum allowed. The minimum height and width for source images is 128. The minimum height and width for destination images depends on the anti-alias setting. If anti-aliasing is disabled, the minimum for both height and width is 128. If 2x1 anti-aliasing is enabled, the height minimum is 128 and the width minimum is 64. If 2x2 anti-aliasing is enabled, the minimum for both height and width is 64.

The *srcformat* and *dstformat* parameters specify the pixel format of the source and destination image. The five types of pixel formats that can be specified are listed in the following table.

**Table 4-22. Pixel Format Specifications**

| Definition | Bits | Description of Format |
|---|---|---|
| GL_RGBA4 | 16 | The R, G, B, and alpha components are 4 bits each. |
| GL_RGB5_A1 | 16 | The R, G, and B components are 5 bits each, and the alpha component is 1 bit. |
| GL_RGB565 | 16 | 5-bit RB components and 6-bit G component. No alpha component. |
| GL_RGB8_OES | 24 | 8-bit RGB components. No alpha component. |
| GL_RGBA8_OES | 32 | The R, G, B, and alpha components are 8 bits each. |

Conversion to a pixel format with a higher pixel depth is not supported. For example, you cannot convert from a 24-bit format to a 32-bit format, or from a 16-bit format to the 24-bit or 32-bit format.

*aamode* specifies the anti-alias filter mode. The three modes that can be specified are listed in the following table. The widths and heights indicate the minimum dimensions of the source image

relative to the destination image.

**Table 4-23. Anti-Aliasing Specifications**

| Definition | Anti-Aliasing | Width | Height |
|---|---|---|---|
| NN_GX_ANTIALIASE_NOT_USED | No anti-aliasing. | Equal | Equal |
| NN_GX_ANTIALIASE_2x1 | Transferred using 2x1 anti-aliasing. | 2 times | Equal |
| NN_GX_ANTIALIASE_2x2 | Transferred using 2x2 anti-aliasing. | 2 times | 2 times |

*yflip* specifies whether vertical flipping is enabled during image transfer. Flipping is performed if GL_TRUE (or a value other than 0) is specified. Flipping is not performed if GL_FALSE (or 0) is specified.

For *blocksize*, specify the block size used for the transfer source image (8 or 32).

**Table 4-24. Errors Generated by the nngxTransferRenderImage() Function**

| Error Code | Cause |
|---|---|
| GL_ERROR_807C_DMP | A command list with object name 0 was bound, or there is no space in the command request queue. |
| GL_ERROR_807D_DMP | Either *srcaddr* or *dstaddr* is not 16-byte aligned. |
| GL_ERROR_807E_DMP | A value other than 8 or 32 is specified in *blocksize*. |
| GL_ERROR_807F_DMP | An invalid value is specified in *aamode*. |
| GL_ERROR_8080_DMP | An invalid value is specified in either *srcformat* or *dstformat*. |
| GL_ERROR_8081_DMP | The pixel size of *srcformat* is greater than that of *dstformat*. |
| GL_ERROR_8082_DMP | An invalid value is specified for *srcwidth*, *srcheight*, *dstwidth*, or *dstheight*. |
| GL_ERROR_8083_DMP | The specified width or height of the destination image is greater than the width or height in pixels of the source image. |
| GL_ERROR_80B7_DMP | The specified height or width of the source image was smaller than the minimum. |
| GL_ERROR_80B8_DMP | The specified height or width of the destination image was smaller than the minimum. |

## 4.1.17. Adding a Linear-to-Block Image Conversion and Transfer Command Request

A command for converting from a linear image to a block image and then transferring the result can be added to the command list by calling the nngxAddL2BTransferCommand() function. (This is one kind of post-filter command request.) The nngxTransferLinearImage() function also provides the same functionality, but the nngxAddL2BTransferCommand() function is more versatile. They also differ in that the latter function adds only a transfer request command and does not add a split command.

**Code 4-23. Function for Adding a Command Request for Converting From a Linear Image to a Block Image and Transferring**

```
void nngxAddL2BTransferCommand(
        const GLvoid* srcaddr, GLvoid* dstaddr,
        GLsizei width, GLsizei height, GLenum format, GLsizei blocksize);
```

*srcaddr* specifies the transfer source (linear image) address. *dstaddr* specifies the transfer destination (block image) address. Both *srcaddr* and *dstaddr* must be 16-byte aligned.

*width* and *height* specify the height and width, in pixels, of the transfer source and transfer destination images. The transfer source and transfer destination images must have the same width and height, and each dimension must be 128 or greater and a multiple of the block size (8 or 32). Finally, if the bit depth of the source image is 24 bits, the image width must be a multiple of 32, even if the block size is 8. The command is not added if `0` is specified for either *width* or *height*.

*format* specifies the pixel format of the image being transferred. The specifiable pixel format is the same as that for the `nngxAddB2LTransferCommand()` function (Table 4-22). The source and destination images must have the same pixel format. Note, however, that if the format is 24-bit, the source image must be in 32-bit format because hardware does not support 24-bit to 24-bit transfer. In this case, the last byte of every 4 bytes of source data is thrown away.

The *blocksize* parameter specifies the block size of the source image as either 8 or 32.

**Table 4-25. Errors Generated by the nngxAddL2BTransferCommand() Function**

| Error Code | Cause |
|---|---|
| GL_ERROR_806F_DMP | A command list with object name 0 was bound or there is no space in the command request queue. |
| GL_ERROR_8070_DMP | Either *srcaddr* or *dstaddr* is not 16-byte aligned. |
| GL_ERROR_8071_DMP | A value other than 8 or 32 is specified in *blocksize*. |
| GL_ERROR_8072_DMP | An invalid value is specified in either *width* or *height*. |
| GL_ERROR_8073_DMP | An invalid value is specified in *format*. |

## 4.1.18. Adding a Block Image Transfer Command Request

A command request for transferring a block image is added to the command list by calling the `nngxAddBlockImageCopyCommand()` function. The added command request allows you to copy graphics between textures and render buffers that contain rendered images. Because transfer is performed by specifying a combination of transfer size and skip size, you can clip part of the source image region or paste to part of the destination image region. The main purpose of this function is to transfer block format images. It can be used for transfer of various types of data because it does not perform format conversion.

**Code 4-24. Function for Adding a Block Image Transfer Command Request**

```
void nngxAddBlockImageCopyCommand(
        const GLvoid* srcaddr, GLsizei srcunit, GLsizei srcinterval,
        GLvoid* dstaddr, GLsizei dstunit, GLsizei dstinterval,
        GLsizei totalsize);
```

Use the *srcaddr* parameter to specify the transfer source start address. *dstaddr* specifies the transfer destination start address. Both *srcaddr* and *dstaddr* must be 16-byte aligned.

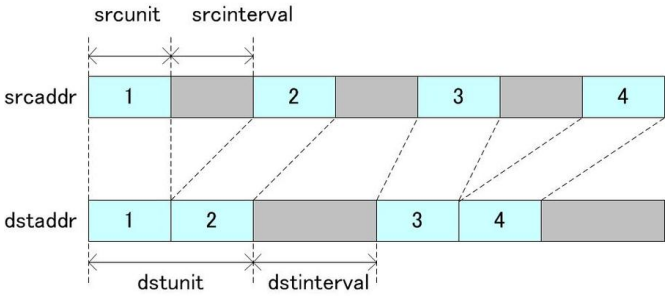
*totalsize* specifies the total amount of data to be transferred, in bytes. *totalsize* must be 16-byte aligned.

*srcunit* and *srcinterval* specify the unit size used for reading each transfer and the skip size, respectively. *srcunit* bytes of data are transferred, and then *srcinterval* bytes in the address being read are skipped, repeating alternately. Transfer ends when the amount of data transferred reaches *totalsize*. If *srcinterval* is `0`, memory is read continuously from the start address until *totalsize* is reached. If *srcinterval* is any value other than `0`, *srcunit* bytes of data are read and then *srcinterval* bytes are skipped, repeatedly. This operation allows part of the

source image to be clipped.

*dstunit* specifies the write unit size of the transfer destination, and *dstinterval* specifies the skip size, in bytes. *dstunit* bytes of data are written and *dstinterval* bytes in the address being written are skipped, repeating alternately. Transfer ends when the amount of data transferred reaches *totalsize*. If *dstinterval* is 0, memory is written continuously from the start address until *totalsize* is reached. If *dstinterval* is any value other than 0, writing and skipping are repeated, allowing the image to be inserted into a portion of the memory region for the transfer destination image.

**Figure 4-3. Sample Block Image Transfer**




The *srcunit*, *srcinterval*, *dstunit*, and *dstinterval* parameters must be multiples of 16. Negative values and values greater than or equal to 0x100000 cannot be specified.

When transferring rendering results, such as block images, note that the start address of the transfer image (at both the source and destination) is normally the upper-left corner of the image (or the lower-left corner in OpenGL ES), and that data is arranged in block units of 8×8 pixels when using a format with a block size of 8. For more information about the block format, see 7.10. Native PICA Format.

**Table 4-26. Errors Generated by the nngxAddBlockImageCopyCommand Function**

| Error Code | Cause |
|---|---|
| GL_ERROR_8074_DMP | A command list with object name 0 was bound or there is no space in the command request queue. |
| GL_ERROR_8075_DMP | Either *srcaddr* or *dstaddr* is not 16-byte aligned. |
| GL_ERROR_8076_DMP | *totalsize* is not a multiple of 16. |
| GL_ERROR_8077_DMP | An invalid value was specified in *srcunit*, *srcinterval*, *dstunit*, or *dstinterval*. |

## 4.1.19. Adding a Memory Fill Command Request

A command request for filling the specified region of memory with the specified data can be added to the command list by calling the nngxAddMemoryFillCommand() function. The command request added by this function can be used for purposes such as clearing the color buffer or depth buffer (stencil buffer). The glClear() function provides the same functionality, but this function is more versatile. Two memory regions of different sizes can be cleared simultaneously by making settings for two channels with independently specifiable parameters.

**Code 4-25. Function for Adding a Memory Fill Command Request**

```
void nngxAddMemoryFillCommand(
        GLvoid* startaddr0, GLsizei size0, GLuint data0, GLsizei width0,
        GLvoid* startaddr1, GLsizei size1, GLuint data1, GLsizei width1);
```

*startaddr0*, *size0*, *data0*, and *width0* represent settings for Channel 0. *startaddr1*, *size1*, *data1*, and *width1* represent settings for Channel 1. Memory is filled simultaneously for both channel 0 and channel 1. If the memory regions specified for Channel 0 and Channel 1 overlap, the fill data that is ultimately applied to the overlapping part is undefined.

*startaddr0* and *startaddr1* specify the start addresses of the memory regions. Addresses must be 16-byte aligned. If 0 is specified for an address, that channel is not used. If **0** is specified for *startaddr0*, no error checking is performed for *size0*, *data0*, or *width0*. If **0** is specified for *startaddr1*, no error checking is performed for *size1*, *data1*, or *width1*.

*size0* and *size1* specify the sizes of the memory regions, in bytes. Sizes must be multiples of 16.

*data0* and *data1* specify the fill pattern data. The specified fill pattern is repeatedly inserted into the memory region until it is full.

*width0* and *width1* specify the bit width of the fill pattern. The values 16, 24, or 32 can be specified for the bit width. If 16 is specified, the memory region is filled in 16-bit units using bits [15:0] of the data. If 24 is specified, the memory region is filled in 24-bit units using bits [23:0] of the data. If 32 is specified, the memory region is filled in 32-bit units using bits [31:0] of the data.

The following table provides fill pattern specifications (bit width and various parameter values) according to the render buffer format being used.

**Table 4-27. Fill Pattern by Render Buffer Format**

| Render Buffer Format | Bit Width | R / D | G / S | B | A |
|---|---|---|---|---|---|
| GL_RGBA8_OES | 32 | [31:24]<br>0 through 255 | [23:16]<br>0 through 255 | [15:8]<br>0 through 255 | [7:0]<br>0 through 255 |
| GL_RGB8_OES | 24 | [23:16]<br>0 through 255 | [15:8]<br>0 through 255 | [7:0]<br>0 through 255 | - |
| GL_RGBA4 | 16 | [15:12]<br>0 through 15 | [11:8]<br>0 through 15 | [7:4]<br>0 through 15 | [3:0]<br>0 through 15 |
| GL_RGB5_A1 | 16 | [15:11]<br>0 through 31 | [10:6]<br>0 through 31 | [5:1]<br>0 through 31 | [0:0]<br>0 through 1 |
| GL_RGB565 | 16 | [15:11]<br>0 through 31 | [10:5]<br>0 through 63 | [4:0]<br>0 through 31 | - |
| GL_DEPTH24_STENCIL8_EXT | 32 | [23:0] | [31:24] | - | - |
| GL_DEPTH_COMPONENT24_OES | 24 | [23:0] | - | - | - |
| GL_DEPTH_COMPONENT16 | 16 | [15:0] | - | - | - |

**Table 4-28. Errors Generated by the nngxAddBlockImageCopyCommand() Function**

| Error Code | Cause |
|---|---|
| GL_ERROR_8078_DMP | A command list with object name **0** was bound, or there is no space in the command request queue. |
| GL_ERROR_8079_DMP | *startaddr0* or *startaddr1* is not 16-byte aligned. |
| GL_ERROR_807A_DMP | *size0* or *size1* is not a multiple of 16. |
| GL_ERROR_807B_DMP | An invalid value is specified in *width0* or *width1*. |

## 4.1.20. Moving the 3D Command Buffer Pointer

Call the nngxMoveCommandbufferPointer() function to move the 3D command buffer pointer

(the position in the 3D commands from which to start running the 3D commands) of the currently bound command list.

**Code 4-26. Function for Moving the 3D Command Buffer Pointer**

```
void nngxMoveCommandbufferPointer(GLint offset);
```

Specify the amount by which to move the pointer **(in bytes)** as the *offset* parameter.

A GL_ERROR_8061_DMP error occurs when no command list is bound, or this operation would move the pointer outside of the 3D command buffer region.

## 4.1.21. Adding Jump Commands

Call the nngxAddJumpCommand() function to add to the currently bound command list a jump command that executes a 3D command in the specified memory region. Use a jump command to move execution to a different command list without causing any interrupts.

This function uses the command buffer execution PICA register. This only uses channel 0, so the content of two registers (0x0238 and 0x023A) are both written when this function is run. For more information, see 8.8.23. Command Buffer Execution Registers (0x0238 – 0x023D) and 8.8.23.1. Consecutive Execution of Command Buffers in *3DS Programming Manual: Advanced Graphics*.

**Code 4-27. Function for Adding Jump Commands**

```
void nngxAddJumpCommand(const GLvoid* bufferaddr, GLsizei buffersize);
```

In *bufferaddr* and *buffersize*, specify the address and size of the command buffer to move execution to. Both *bufferaddr* and *buffersize* must be multiples of 16.

The content of the destination command buffer (the command list specified by *bufferaddr* and *buffersize*) is not copied to the command buffer of the currently bound command list. A jump command changes the execution address of a command buffer and directly executes the destination command buffer. Consequently, the application must ensure that the jump destination memory cache has been flushed.

The last command executed at the jump destination must be a split command (a command to write to the split command setting register, added by the nngxSplitDrawCmdlist() function). Alternatively, this command could be another jump command. When using multiple jump commands, the last command in the last command buffer in the chain must be a split command.

This function adds a command request for a 3D execution command. A GL_ERROR_809A_DMP error occurs when this function is called immediately after the command buffer has been flushed (for example, by a call to the nngxFlush3DCommand() function) because doing so is meaningless. To add a 3D command to the command buffer immediately after a flush, call the nngxAdd3DCommand() function.

**Table 4-29. Errors Generated by the nngxAddJumpCommand() Function**

| Error Code | Cause |
|---|---|
| GL_ERROR_8096_DMP | The bound command list's object name is 0. |
| GL_ERROR_8097_DMP | *buffersize* is 0 or less. |
| GL_ERROR_8098_DMP | *buffersize* is not a multiple of 16. |
| GL_ERROR_8099_DMP | *bufferaddr* is not a multiple of 16. |
| GL_ERROR_809A_DMP | This function was called immediately after the command buffer was flushed. |

| | |
|---|---|
| GL_ERROR_809B_DMP | The command request added by this function makes the queue overflow. |
| GL_ERROR_809C_DMP | The command added by this function makes the command buffer overflow. |

## 4.1.22. Adding Subroutine Commands

Call the nngxAddSubroutineCommand() function to add both a jump command to execute a 3D command in the specified memory region and a command to set the address for returning to the command buffer jumped from, to the currently bound command list. Use a subroutine command to execute another command list without causing any interrupts, as if it were a subroutine.

This function uses the command buffer execution PICA register. This uses all channels, so the content of four registers (0x0238 through 0x023B) are written when this function is run. For more information, see 8.8.23. Command Buffer Execution Registers (0x0238 – 0x023D) and 8.8.23.1. Consecutive Execution of Command Buffers in *3DS Programming Manual: Advanced Graphics*.

**Code 4-28. Function for Adding Subroutine Commands**

```
void nngxAddSubroutineCommand(const GLvoid* bufferaddr, GLsizei buffersize);
```

In *bufferaddr* and *buffersize*, specify the address and size of the command buffer to move execution to. Both *bufferaddr* and *buffersize* must be multiples of 16.

The content of the destination command buffer (the command list specified by *bufferaddr* and *buffersize*) is not copied to the command buffer of the currently bound command list. A jump command changes the execution address of a command buffer and directly executes the destination command buffer. Consequently, the application must ensure that the jump destination memory cache has been flushed.

The jump command is executed on channel 0, and the command to return to the command buffer jumped from is executed on channel 1. Consequently, the last command executed at the jump destination must be a kick command for channel 1 (a command to write to the command buffer execution register 0x023D). Alternatively, this command could be a jump command to another command buffer, but the channel used by the jump must not be channel 0, and the last command in the last command buffer in the chain must be a kick command for channel 1. In addition, you must not write to the channel 1 address setting registers (0x0239 and 0x023B). This function adds a jump command (channel 0) and an address setting (channel 1). The application must place the channel 1 kick command and the jump commands within the subroutine.

This function does not add a command request for a 3D execution command. After calling this function, continue accumulating commands, and then execute them after flushing the command buffer, such as by using the nngxFlush3DCommand() function. Values written to the channel 1 size setting register (0x023B) added by this function are undefined until the command buffer is flushed. Operation is similarly undefined if you reuse the copied content of this register until the command buffer is flushed.

**Table 4-30. Errors Generated by the nngxAddJumpCommand() Function**

| Error Code | Cause |
|---|---|
| GL_ERROR_809D_DMP | The bound command list's object name is 0. |
| GL_ERROR_809E_DMP | *buffersize* is 0 or less. |
| GL_ERROR_809F_DMP | *buffersize* is not a multiple of 16. |
| GL_ERROR_80A0_DMP | *bufferaddr* is not a multiple of 16. |
| GL_ERROR_80A1_DMP | The command added by this function makes the command buffer overflow. |

# 4.2. Command Request Types

The following command requests are queued in a command list.

### DMA Transfer Command Requests

These command requests use DMA transfers to send texture images and vertex buffers from main memory into VRAM.

These command requests are queued by `glTexImage2D()` and other functions that allocate texture regions, and by `glBufferData()` and other functions that allocate vertex buffer regions.

### Render Command Requests

These command requests execute a single command set of 3D commands accumulated in the 3D command buffer.

When `glClear()`, `glTexImage2D()`, and other functions are called, they write a *buffer loading complete* 3D command and then queue the accumulated 3D command buffer as a single render command request.

The `nngxSplitDrawCmdlist()` function allows you to queue render command requests at any time.

### Memory-Fill Command Requests

These command requests use the GPU memory-fill feature to clear a region allocated in VRAM using a specified data pattern.

These command requests specify a render buffer and are queued when the `glClear()` function is called. The `glClear()` function also requires a 3D command other than a memory-fill command request to be executed. In other words, when the `glClear()` function is called, it first writes 3D commands for the `glClear()` function and a buffer loading complete 3D command, and then it queues a render command request and a memory-fill command request.

### Post-Transfer Command Requests

These command requests use the GPU post-filter feature to convert images rendered in PICA block format into a linear format that can be read by the LCDs.

These command requests are queued when the `nngxTransferRenderImage()` function is called. If the `nngxSplitDrawCmdlist()` function has not been called in advance to stop reading from the 3D command buffer, these command requests are queued after a buffer loading complete command is written and a render command request is queued.

### Copy Texture Command Requests

These command requests copy GPU rendering results into memory as texture images.

These command requests are queued when `glCopyTexImage2D()` or `glCopyTexSubImage2D()` is called.

If the `nngxSplitDrawCmdlist()` function has not been called in advance to stop reading from the 3D command buffer, these command requests are queued after a buffer loading complete command is written and a render command request is queued.

# 4.3. Methods for Optimizing 3D Command Buffer Performance

The following information describes methods for optimizing performance during 3D command buffer execution.

## 4.3.1. Changes in Load Speed due to Address and Size

The address and size of a 3D command buffer can have an effect on load speed at run time.

There are two types of command buffer execution: executing 3D execution commands queued in a command request, and executing the command buffer execution register.

When executing 3D execution commands, execution is affected by the size from the address immediately after a split command added by `nngxFlush3DCommand()` or `nngxSplitDrawCmdlist()`, up to the next split command added. You can get the address of 3D commands being accumulated in the 3D command buffer by calling the `nngxGetCmdlistParameteri()` function and passing `NN_GX_CMDLIST_CURRENT_BUFADDR` for *`pname`*.

When executing using the command buffer execution register, execution is affected by the address and size of the following command buffers: added by `nngxAddJumpCommand()`, added as subroutines by `nngxAddSubroutineCommand()`, or executed to return from a subroutine to the calling location.

If the 3D command buffer address is 128-byte aligned, and the size is a multiple of 256 bytes (256, 512, 768, and so on), transfer speed may be faster.

If the 3D command buffer address is not 128-byte aligned and the size starting from the previous 128-byte aligned address to the end of the 3D command buffer is a multiple of 256, speed may be increased. For example, if the 3D command buffer address and size are `0x20000010` and `0x1F0` respectively, the preceding 128-byte aligned address is only `0x10` earlier, at `0x20000000`. The distance from there to the end is `0x1F0 + 0x10`, which is `0x200` (and a multiple of 256).

Although the address and size of the 3D command buffer can influence loading speed as just described because of the way the GPU is implemented, you may not see large benefits due to factors such as the location of the buffer, the content of 3D commands, and memory access conflicts with other modules.

## 4.3.2. Using Subroutine Execution

It may be possible to improve performance by using 3D command buffer subroutine execution.

### 4.3.2.1. Overview

3D command buffer subroutine execution uses the command buffer execution register for execution. In contrast to the ordinary method of storing 3D commands in a sequence of 3D command buffers and executing it, a command buffer stored in a different location is executed successively using a command buffer address jump feature. This method is called command buffer subroutine execution because of performing the following controls: first performing an address jump specifying the address of a 3D command buffer, executing the 3D command buffer at that location, and then returning to the calling location.

For more information about using command buffer subroutine execution, see 4.1.22. Adding Subroutine Commands and Command Buffer Execution Registers in the *3DS Programming Manual: Advanced Graphics*.

## 4.3.2.2. Effect on Behavior

Command buffer subroutine execution has the following advantages.

- Only a jump command to the subroutine command buffer needs to be stored, eliminating the CPU processing needed to copy the 3D commands. The technique is effective for tasks that are quite large and configured frequently, such as loading reference table data or shader programs.
- The subroutine command buffer is not copied to the current 3D command buffer, but is referenced directly by the GPU, allowing the total size of the command buffer to be reduced.
- If the subroutine command buffer is stored in VRAM, GPU access to the command buffer is faster than if it is in main memory (device memory). If memory access to the command buffer is a performance bottleneck, this technique could improve overall system processing speed.

On the other hand, it has the following disadvantage.

- Switching the address due to a jump command incurs memory access overhead. If the granularity of subroutines in the implementation is small and they are called frequently, a decrease in GPU processing speed could result.

The effect of converting to subroutines on processing performance is heavily influenced by issues such as memory access conflicts, so it is strongly dependent on the actual implementation of the application.

## 4.3.2.3. Storage Location

Command buffer access speed is faster in VRAM than in main memory (device memory), so we recommend storing subroutine command buffers in VRAM.

There is some memory access overhead when executing a subroutine command buffer using a jump command, but if the executed command buffer is stored in VRAM, this overhead is decreased.

To store a command buffer in VRAM, it must first be generated in device memory and then transferred to VRAM by DMA using `nngxAddVramDmaCommand()`. For information about DMA transfers to VRAM, see 4.1.13. Adding a DMA Transfer Command Request.

## 4.3.2.4. Balance Between Execution and Access Processes

Depending on the content of subroutine command buffers, the processing bottleneck could move between accessing and executing 3D commands.

If the 3D command is the register write command of the rasterization module or a later module (including the rasterization module), each 3D command requires 2 cycles to process, so it is relatively processor-intensive. When 3D commands are composed of burst commands, execution

is even more processor-intensive relative to access processing. In this case, the bottleneck is in command execution, and the processing cost of memory access due to conversion to subroutines is hidden.

If the 3D command is the register write command of a module before the rasterization module (not including the rasterization module), each 3D command requires only one cycle to process, so processing emphasis is light relative to commands discussed in the previous paragraph. In this case, the bottleneck is more likely to be access processing, and the memory access processing cost incurred by conversion to subroutines is more likely to affect the overall performance.

For information about the relative positioning of each module, see 2.2. Rendering Pipeline.



# 5. Shader Programs

Shader programs can customize the 3D graphics pipeline and control various graphics effects on the 3DS system.

There are three types of 3DS shader programs: one processes vertices, one creates geometry, and one processes fragments.

The shader program that processes vertices (the vertex shader) can be a unique shader programmed by developers.

The shader program that creates geometry (the geometry shader) is provided by the SDK. Geometry shaders and vertex shaders can be used in conjunction with each other.

The shader program that processes fragments (the fragment shader) is not programmable. Fragment processing is implemented as a fixed pipeline, but it can be controlled through reserved uniforms. This document uses the terms *reserved fragment processing* and *reserved fragment shader* to refer to the fragment pipeline and shader program, respectively.

## 5.1. Creating Shaders

Vertex shaders are the only shader programs that can be created by developers. The series of procedures related to vertex processing follow the OpenGL ES 2.0 specifications, but some features are not supported.

Shader programs are written in an assembly language that is unique to the PICA graphics core. For an application to use a shader program, it must load and then attach a binary generated by a special-purpose assembler and linker. The `glShaderSource()` and `glCompileShader()` functions in the OpenGL ES 2.0 specifications are not implemented.

For more information about how to create a vertex shader, see 8. Vertex Shader and the Vertex Shader Reference.

## 5.2. Loading Shaders

As explained in 5.1. Creating Shaders, you must load and then attach a shader program's binary

data. Use the `glCreateShader()` function first to create a shader object.

**Code 5-1. Definition of the glCreateShader Function**

```
GLuint glCreateShader(GLenum type);
```

The value to pass as *`type`* depends on the shader program being loaded. The reserved fragment shader uses a fixed implementation and does not need to be loaded.

**Table 5-1. Shader Object Types**

| Type | Generated Object |
|---|---|
| GL_VERTEX_SHADER | A developer-created vertex shader object. |
| GL_GEOMETRY_SHADER_DMP | A geometry shader object provided by the SDK. |

Load the shader program binary into memory and then bind it to the GPU with the `glShaderBinary()` function.

**Code 5-2. Definition of the glShaderBinary Function**

```
void glShaderBinary(GLint n, const GLuint* shaders, GLenum binaryformat,
                    const void* binary, GLint length);
```

For *`shaders`*, specify an array of shader objects, and for *`n`*, specify the number of array elements. Because you can only load a binary that was created by the assembler and linker, set *`binaryformat`* to GL_PLATFORM_BINARY_DMP. Set *`binary`* to the address at which the shader program binary was loaded and set *`length`* to the binary's size (in bytes).

The loaded shader programs are bound to the array in the order that they were passed to the linker. You can use the map file output by the linker to check the number of elements in the array and the type of shader objects to set. For more information, see the Vertex Shader Reference.

## 5.3. Attaching Shaders

To use a loaded shader program, an application must attach the shader program to a program object created by `glCreateProgram`, and then link that program object.

**Code 5-3. Definition of the glCreateProgram Function**

```
GLuint glCreateProgram(void);
```

Unlike in OpenGL ES 2.0, program objects have a 13-bit namespace that is independent of shader objects. Consequently, up to 8191 objects can be created simultaneously. If any of these objects are deleted by `glDeleteProgram`, they can be re-created.

Use the `glAttachShader()` function to attach a shader program to a program object.

**Code 5-4. Definition of the glAttachShader Function**

```
void glAttachShader(GLuint program, GLuint shader);
```

Set *program* to the return value from `glCreateProgram()` and set *shader* to the shader object.

Both the vertex shader and geometry shader are attached from loaded binaries. The reserved fragment shader, on the other hand, does not need to be loaded and is attached by setting *shader* to `GL_DMP_FRAGMENT_SHADER_DMP`.

You can attach one vertex shader, one geometry shader, and one reserved fragment shader to a single program object. In other words, if a point shader and line shader (both geometry shaders) are attached one after the other, only the line shader is activated.

Use the `glLinkProgram()` function to link a program object.

**Code 5-5. Definition of the glLinkProgram Function**

```
void glLinkProgram(GLuint program);
```

You can link more than one program object. However, if you use `glAttachShader` to attach a different shader program to a linked program object, you must use `glLinkProgram` to relink the program object. The program object fails to be linked when the vertex shader and geometry shader use a total of more than 2048 uniforms.

## 5.4. Using Shaders

Call the `glUseProgram()` function to apply a linked shader program to the 3D processing pipeline.

**Code 5-6. Definition of the glUseProgram Function**

```
void glUseProgram(GLuint program);
```

This function allows you to switch between several linked shader programs.

OpenGL allowed you to call the `glValidateProgram()` function to validate a shader program, but this function does nothing when it is called on a 3DS system.

**Code 5-7. Definition of the glValidateProgram Function**

```
void glValidateProgram(GLuint program);
```

## 5.5. Detaching Shaders

You can use the `glDetachShader()` function to detach shader programs that are no longer necessary.

**Code 5-8. Definition of the glDetachShader Function**

```
void glDetachShader(GLuint program, GLuint shader);
```

## 5.6. Destroying Shaders

You can use the `glDeleteShader()` function to destroy shader objects that are no longer necessary.

**Code 5-9. Definition of the glDeleteShader Function**

```
void glDeleteShader(GLuint shader);
```

## 5.7. Querying Shaders

You can query program objects and shader objects to determine whether they are valid or invalid, and to get parameters and other shader-related information.

### 5.7.1. Validation

You can use the `glIsProgram()` and `glIsShader()` functions to determine whether program objects and shader objects are valid.

**Code 5-10. Definition of the glIsProgram and glIsShader Functions**

```
GLboolean glIsProgram(GLuint program);
GLboolean glIsShader(GLuint shader);
```

These functions return a value of `GL_TRUE` if the program object or shader object passed as an argument is valid and `GL_FALSE` if it is not.

### 5.7.2. Getting Attached Shader Objects

You can use the `glGetAttachedShaders()` function to get the shader objects attached to a program object.

**Code 5-11. Definition of the glGetAttachedShaders Function**

```
void glGetAttachedShaders(GLuint program, GLsizei maxcount, GLsizei* count,
                          GLuint* shaders);
```

A list of shader objects attached to the program object specified by *`program`* is stored in the array specified by *`shaders`*. For *`maxcount`*, specify the size of the array specified in *`shaders`*. A `GL_INVALID_VALUE` error is generated if *`program`* is invalid or *`maxcount`* is negative.

The *`count`* parameter holds the number of shader objects that were saved. This value is not saved if `NULL` is specified, but you can get the number of attached shader objects by calling the `glGetProgramiv()` function (described later) with `GL_ATTACHED_SHADERS` passed as an argument.

## 5.7.3. Getting Command List Parameters

You can use the `glGetProgramiv()` and `glGetShaderiv()` functions to get parameters for program objects and shader objects.

**Code 5-12. Definition of the glGetProgramiv and glGetShaderiv Functions**

```
void glGetProgramiv(GLuint program, GLenum pname, GLint* params);
void glGetShaderiv(GLuint shader, GLenum pname, GLint* params);
```

These functions store parameter values in *params* that correspond to the parameter name specified by *pname*. A `GL_INVALID_ENUM` error is generated if an invalid value is specified for *pname*. A `GL_INVALID_VALUE` error is generated if an invalid value is specified for *program* or *shader*.

The following table shows the parameter names that can be specified for *pname* and the parameters that are stored in *params* for the `glGetProgramiv()` function.

**Table 5-2. Specifiable Parameter Names and Values Saved for the glGetProgramiv Function**

| pname | Values Saved in params |
|---|---|
| `GL_DELETE_STATUS` | `GL_TRUE` if the program object is in the "waiting to be deleted" state, and `GL_FALSE` otherwise. A program object in use by `glUseProgram` transitions to the "waiting to be deleted" state when `glDeleteProgram` is called on it. |
| `GL_LINK_STATUS` | `GL_TRUE` if `glLinkProgram` has successfully linked the program object, and `GL_FALSE` otherwise. |
| `GL_VALIDATE_STATUS` | The same value as when `GL_LINK_STATUS` is specified. |
| `GL_INFO_LOG_LENGTH` | Always `0`. |
| `GL_ATTACHED_SHADERS` | The number of shader objects attached to the program object. |
| `GL_ACTIVE_ATTRIBUTES` | The number of vertex attributes in the active state. |
| `GL_ACTIVE_ATTRIBUTE_MAX_LENGTH` | The number of characters in the longest name of the active vertex attributes. The character count includes the terminating character (`NULL`). |
| `GL_ACTIVE_UNIFORMS` | The number of uniforms in the active state. |
| `GL_ACTIVE_UNIFORM_MAX_LENGTH` | The number of characters in the longest name of the active uniforms. The character count includes the terminating character (`NULL`). |

The following table shows the parameter names that can be specified for *pname*, and the parameters that are stored in *params* for the `glGetShaderiv()` function.

**Table 5-3. Specifiable Parameter Names and Values Saved for the glGetShaderiv Function**

| pname | Values Saved in params |
|---|---|
| `GL_SHADER_TYPE` | The shader type. |
| `GL_DELETE_STATUS` | `GL_TRUE` if the shader object is in the "waiting to be deleted" state, and `GL_FALSE` otherwise. The shader object attached to a program object transitions to the "waiting to be deleted" state when `glDeleteShader` is called on it. |
| `GL_COMPILE_STATUS` | Always `GL_FALSE`. |
| `GL_INFO_LOG_LENGTH` | Always `0`. |
| `GL_SHADER_SOURCE_LENGTH` | Always `0`. |

# 6. Vertex Buffers

Vertex buffers store vertex coordinates, colors, and indices, in addition to texture coordinates and other information. Use vertex buffers to process certain kinds of models, such as those with a large number of vertices, in the vertex shader. If you do not use vertex buffers, heavy CPU processing (such as vertex array sorting) could cause a considerable drop in performance.

## 6.1. Creating Objects

Use the `glGenBuffers()` function to create the buffer objects.

**Code 6-1. Definition of the glGenBuffers Function**

```
void glGenBuffers(GLsizei n, GLuint* buffers);
```

This code creates *n* buffer objects and stores their object names in *buffers*.

## 6.2. Specifying Objects

Specify buffer objects to bind as vertex buffers with the `glBindBuffer()` function. After this function is called, processing for each type of vertex buffer is run on the specified buffer objects.

**Code 6-2. Definition of the glBindBuffer Function**

```
void glBindBuffer(GLenum target, GLuint buffer);
```

Set *target* to the vertex buffer type. *buffer* specifies the buffer object created by the `glGenBuffers()` function. A buffer object is created for an object name as long as that uncreated object name is specified for *buffer*.

**Table 6-1. Vertex Buffer Types**

| target Value | Vertex Buffer Types |
|---|---|
| `GL_ARRAY_BUFFER` | Buffer for vertex coordinates, vertex colors, normals, and so on. |
| `GL_ELEMENT_ARRAY_BUFFER` | Index buffer used by the `glDrawElements()` function. |
| `GL_VERTEX_STATE_COLLECTION_DMP` | Vertex state collection (see 6.6. Vertex State Collections). |

## 6.3. Allocating Buffers

Use the `glBufferData()` function to allocate a buffer region, and then load vertex data.

**Code 6-3. Definition of the glBufferData Function**

```
void glBufferData(GLenum target, GLsizeiptr size, const void* data,
                  GLenum usage);
```

For *`target`* specify the same value specified in the `glBindBuffer()` function (Table 6-1).

For *`data`* and *`size`*, specify the vertex data to store and its size, respectively. When *`data`* is `0` (`NULL`) the region is simply allocated and no data is stored.

The value of *`usage`* must be `GL_STATIC_DRAW`.

> **Note:** To configure the GPU access targets and processing to use when allocating the buffer region, pass a bitwise OR of some specific flag values for `target` for the `glBufferData()` function. For more information, see 3. Using Data Located in Main Memory in *3DS Programming Manual: Advanced Graphics*.

## 6.4. Rewriting Buffers

Use the `glBufferSubData()` function to rewrite part of the buffer that was allocated by `glBufferData`.

**Code 6-4. Definition of the glBufferSubData Function**

```
void glBufferSubData(GLenum target, GLintptr offset, GLsizeiptr size,
                     const void* data);
```

For *`target`* specify the same value specified in the `glBindBuffer()` function (Table 6-1).

For *`offset`*, specify the offset to the section to rewrite.

For *`data`* and *`size`*, specify the data to write and its size, respectively.

## 6.5. Freeing Buffers

Use the `glDeleteBuffers()` function to destroy buffer objects that are no longer necessary.

**Code 6-5. Definition of the glDeleteBuffers Function**

```
void glDeleteBuffers(GLsizei n, const GLuint* buffers);
```

This destroys the buffer objects specified by the n object names in *`buffers`*.

## 6.6. Vertex State Collections

Vertex state collections are new to the Nintendo 3DS. You can bind buffer objects to vertex buffers and record them. By using vertex state collections, you can bind buffer objects to vertex buffers and set vertex attributes for a group of vertices.

Vertex state collections share a namespace with buffer objects and can be created, specified, and destroyed using `glGenBuffers`, `glBindBuffer`, and `glDeleteBuffers`, respectively.

### 6.6.1. Creating Vertex State Collections

A vertex state collection operates as a special buffer object. Use the `glGenBuffers()` function to generate objects that can be used as vertex state collections, just as you would for buffer objects.

### 6.6.2. Specifying Vertex State Collections

Use the `glBindBuffer()` function to specify a buffer object to use as a vertex state collection. For *`target`*, specify `GL_VERTEX_STATE_COLLECTION_DMP`. By default, the object with a name of `0` (with a value of `0` passed into *`buffer`*) is a vertex state collection.

After this function is called, the vertex state collection records which buffer objects are bound by `glBindBuffer` as vertex buffers (using `GL_ARRAY_BUFFER` or `GL_ELEMENT_ARRAY_BUFFER`), and also records which vertex attribute values are set by `glEnableVertexAttribArray`, `glDisableVertexAttribArray`, `glVertexAttrib{1234}{fv}`, or `glVertexAttribPointer`. When a buffer object is bound as a vertex buffer, it overwrites any other existing bindings to the same vertex buffer target. Settings continue to be recorded in a vertex state collection until it is switched.

When the vertex state collection is switched, all buffer objects recorded in the new vertex state collection are bound as the new vertex buffers.

### 6.6.3. Destroying Vertex State Collections

As with a buffer object, you can use the `glDeleteBuffers()` function to destroy vertex state collections. Even if a vertex state collection is destroyed, it does not affect the binding between the vertex buffer target and the buffer objects recorded in that vertex state collection.

The `glDeleteBuffers()` function does not immediately destroy a vertex state collection that is in use. A vertex state collection remains in use until it is switched with another one. You cannot destroy the default vertex state collection. Calls to `glDeleteBuffers` on the default vertex state collection are ignored.

## 6.7. Sample Vertex Buffer Usage

The following code samples show how to render a single triangle using a vertex buffer and the

`glDrawElements()` function. There are three steps: defining the arrays, allocating the buffers, and rendering the triangle.

**Code 6-6. Defining the Arrays**

```
GLuint triIndexID, triArrayID;
GLushort triIndex[1 * 3] = { 0, 1, 2 };
GLfloat triVertex[3 * 4] =
{
     0.5f,  0.0f, 0.0f, 1.0f,
    -0.5f,  0.5f, 0.0f, 1.0f,
    -0.5f, -0.5f, 0.0f, 1.0f
};
GLfloat triVertexColor[3 * 3] =
{
    1.0f, 0.0f, 0.0f,
    0.0f, 1.0f, 0.0f,
    0.0f, 0.0f, 1.0f
};
```

**Code 6-7. Allocating the Buffers**

```
glGenBuffers(1, &triArrayID);
glBindBuffer(GL_ARRAY_BUFFER, triArrayID);
glBufferData(GL_ARRAY_BUFFER, sizeof(triVertex) + sizeof(triVertexColor), 0,
        GL_STATIC_DRAW);
glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(triVertex), triVertex);
glBufferSubData(GL_ARRAY_BUFFER, sizeof(triVertex), sizeof(triVertexColor),
        triVertexColor);
glGenBuffers(1, &triIndexID);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, triIndexID);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, 3 * sizeof(GLushort), triIndex,
GL_STATIC_DRAW);
```

**Code 6-8. Rendering With glDrawElements**

```
glBindBuffer(GL_ARRAY_BUFFER, triArrayID);
glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 0, 0);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, (GLvoid*)sizeof(triVertex));

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, triIndexID);
glDrawElements(GL_TRIANGLES, 3, GL_UNSIGNED_SHORT, 0);
```

## 6.8. Restrictions on Vertex Data Placement

The following hardware restrictions apply to the vertex data placement configured by the `glVertexAttribPointer()` function when rendering using vertex data stored in a vertex buffer. A `GL_INVALID_OPERATION` error is generated when the `glDrawArrays()` or `glDrawElements()` function is called in violation of these restrictions.

- All vertex data must be aligned to its own data type's size.
- The stride for all vertex data in a single structure must be a multiple of the size of that structure's largest vertex data type.
- If any padding (beyond the minimum amount required to meet the two aforementioned restrictions) is inserted after the vertex data, it must start at the end of the vertex data and continue until the second-closest 4-byte boundary.

The compiler automatically inserts padding, so there will be no conflicts with the first two restrictions

even if you do not pay attention to them when coding.

The following sample code would conflict with the last restriction, if it did not insert `extraPadding2`.

**Code 6-9. Sample Code for Working Around Restrictions on Vertex Data Placement**

```
struct tagVertex
{
    GLshort position[3];
    GLshort extraPadding1;
    GLshort extraPadding2[2];
    GLshort color[4];
};
```

You cannot mix vertex attributes and indices that use vertex buffers with attributes and indices that do not use them in a single call to the `glDrawArrays` or `glDrawElements()` function. A `GL_INVALID_OPERATION` error is generated if they are mixed together.

### 6.8.1. Restrictions Affecting Only glDrawElements

A `GL_INVALID_OPERATION` error is generated when all of the following conditions are met, due to restrictions associated with the method of vertex array storage.

- The vertex buffer is being used.
- There are 12 vertex attributes being used.
- All vertex attributes are being used as vertex arrays. (`qlEnableVertexAttribArray` has been called for all vertex attributes.)
- Rendering is performed using `qlDrawElements`.

When all these conditions are present, at least two vertex attributes must be placed as interleaved arrays. In other words, 12 independent arrays cannot be simultaneously used as vertex attributes.

> **Note:** An interleaved array is a group made up of multiple vertex attributes defined as a structure. An individual array made up of one vertex attribute is called an independent array.



# 7. Textures

In addition to cube map textures and 2D textures that are pasted onto polygon models, the 3DS system can handle shadow textures used for shadows, gas textures used for rendering gases, and lookup tables used by the reserved fragment shader (these lookup tables cannot be used as textures).

This chapter explains the necessary procedures for using these textures and describes how the native PICA format differs from the OpenGL specifications.

## 7.1. Creating Texture Objects

Use the `glGenTextures()` function to create texture objects to bind as textures.

**Code 7-1. Definition of the glGenTextures Function**

```
void glGenTextures(GLsizei n, GLuint* textures);
```

## 7.2. Specifying Texture Objects

Use the `glBindTexture()` function to specify a texture object to bind as a texture. After this function is called, the various texture processes are performed using the specified texture object. Texture images loaded for (and the results of processing performed on) a texture object are preserved until the texture object is deleted. Consequently, you can switch texture objects to change textures without reloading texture images.

**Code 7-2. Definition of the glBindTexture Function**

```
void glBindTexture(GLenum target, GLuint texture);
```

Specify the texture type for *`target`*, and the texture object to bind for *`texture`*. A `GL_INVALID_ENUM` error is generated if you set *`target`* to a value that is not in the following table.

**Table 7-1. Texture Types**

| target Value | Type of Texture |
|---|---|
| `GL_TEXTURE_2D` | 2D textures, shadow textures, and gas textures. |
| `GL_TEXTURE_CUBE_MAP_POSITIVE_X` | Cube-map texture. |
| `GL_TEXTURE_CUBE_MAP_NEGATIVE_X` | Cube-map texture. |
| `GL_TEXTURE_CUBE_MAP_POSITIVE_Y` | Cube-map texture. |
| `GL_TEXTURE_CUBE_MAP_NEGATIVE_Y` | Cube-map texture. |
| `GL_TEXTURE_CUBE_MAP_POSITIVE_Z` | Cube-map texture. |
| `GL_TEXTURE_CUBE_MAP_NEGATIVE_Z` | Cube-map texture. |
| `GL_LUT_TEXTURE`*`i`*`_DMP` (where *`i`* is a value between 0 and 31) | Lookup tables (used by the reserved fragment shader). |
| `GL_TEXTURE_COLLECTION_DMP` | Texture collections (see 7.9. Texture Collections). |

## 7.3. Loading Texture Images

In addition to normal texture images (2D textures), the `glTexImage2D()` function loads the following special textures: cube map textures, shadow textures, and gas textures. **Partial texture loading with**

**the `glTexSubImage2D()` function is not supported.**

**Code 7-3. Definition of the glTexImage2D Function**

```
void glTexImage2D(GLenum target, GLint level, GLenum internalformat,
                  GLsizei width, GLsizei height, GLint border, GLenum format,
                  GLenum type, const void* pixels);
```

Set *`target`* to one of the following values.

**Table 7-2 target Values in glTexImage2D**

| target Value | Texture Usage |
|---|---|
| GL_TEXTURE_2D | 2D textures, shadow textures, and gas textures. |
| GL_TEXTURE_CUBE_MAP_POSITIVE_X | Cube map +X plane textures. |
| GL_TEXTURE_CUBE_MAP_NEGATIVE_X | Cube map -X plane textures. |
| GL_TEXTURE_CUBE_MAP_POSITIVE_Y | Cube map +Y plane textures. |
| GL_TEXTURE_CUBE_MAP_NEGATIVE_Y | Cube map −Y plane textures. |
| GL_TEXTURE_CUBE_MAP_POSITIVE_Z | Cube map +Z plane textures. |
| GL_TEXTURE_CUBE_MAP_NEGATIVE_Z | Cube map −Z plane textures. |

Combinations of *`format`* and *`type`* specify the format of texture images stored in the region specified by *`pixels`*. The following table lists the formats that can be handled by the 3DS system. The **Bytes** column indicates the number of bytes per texel. An asterisk (*) that follows the byte count indicates a native PICA format, which has a different byte order than the standard OpenGL format. For more information about the native PICA format, see 7.10. Native PICA Format.

**Table 7-3. Texture Image Formats Specified by format and type**

| format | type | Format | Bytes |
|---|---|---|---|
| GL_RGBA | GL_UNSIGNED_SHORT_4_4_4_4 | RGBA4 | 2 |
| GL_RGBA | GL_UNSIGNED_SHORT_5_5_5_1 | RGBA5551 | 2 |
| GL_RGBA | GL_UNSIGNED_BYTE | RGBA8 | 4 |
| GL_RGB | GL_UNSIGNED_SHORT_5_6_5 | RGB565 | 2 |
| GL_RGB | GL_UNSIGNED_BYTE | RGB8 | 3 |
| GL_ALPHA | GL_UNSIGNED_BYTE | A8 | 1 |
| GL_ALPHA | GL_UNSIGNED_4BITS_DMP | A4 | 0.5 |
| GL_LUMINANCE | GL_UNSIGNED_BYTE | L8 | 1 |
| GL_LUMINANCE | GL_UNSIGNED_4BITS_DMP | L4 | 0.5 |
| GL_LUMINANCE_ALPHA | GL_UNSIGNED_BYTE | LA8 | 2 |
| GL_LUMINANCE_ALPHA | GL_UNSIGNED_BYTE_4_4_DMP | LA4 | 1 |
| GL_SHADOW_DMP | GL_UNSIGNED_INT | - | 4 |
| GL_GAS_DMP | GL_UNSIGNED_SHORT | - | 4 |
| GL_HILO8_DMP | GL_UNSIGNED_BYTE | - | 2 |
| GL_RGBA_NATIVE_DMP | GL_UNSIGNED_SHORT_4_4_4_4 | RGBA4 | 2 |
| GL_RGBA_NATIVE_DMP | GL_UNSIGNED_SHORT_5_5_5_1 | RGBA5551 | 2 |
| GL_RGBA_NATIVE_DMP | GL_UNSIGNED_BYTE | RGBA8 | 4* |
| GL_RGB_NATIVE_DMP | GL_UNSIGNED_SHORT_5_6_5 | RGB565 | 2 |
| GL_RGB_NATIVE_DMP | GL_UNSIGNED_BYTE | RGB8 | 3* |

| | | | |
|---|---|---|---|
| GL_ALPHA_NATIVE_DMP | GL_UNSIGNED_BYTE | A8 | 1 |
| GL_ALPHA_NATIVE_DMP | GL_UNSIGNED_4BITS_DMP | A4 | 0.5 |
| GL_LUMINANCE_NATIVE_DMP | GL_UNSIGNED_BYTE | L8 | 1 |
| GL_LUMINANCE_NATIVE_DMP | GL_UNSIGNED_4BITS_DMP | L4 | 0.5 |
| GL_LUMINANCE_ALPHA_NATIVE_DMP | GL_UNSIGNED_BYTE | LA8 | 2* |
| GL_LUMINANCE_ALPHA_NATIVE_DMP | GL_UNSIGNED_BYTE_4_4_DMP | LA4 | 1 |
| GL_SHADOW_NATIVE_DMP | GL_UNSIGNED_INT | - | 4* |
| GL_GAS_NATIVE_DMP | GL_UNSIGNED_SHORT | - | 4* |
| GL_HILO8_DMP_NATIVE_DMP | GL_UNSIGNED_BYTE | - | 2* |

R, G, and B represent colors (red, green, and blue, respectively).
A represents the alpha value.
L represents the luminance value.

As set forth in the OpenGL ES 1.1 specifications, a value of 1 is output for the alpha component if the texture combiner references a texture without an alpha component. This is also true for compressed textures.

You can only set *format* to GL_RGB or GL_RGB_NATIVE_DMP and *type* to GL_UNSIGNED_BYTE when *target* is GL_TEXTURE_2D. When *format* is GL_∗_NATIVE_DMP, *pixels* must specify data in the native PICA format. When GL_GAS_DMP is specified, *pixels* must be 0 (NULL).

For *width* and *height*, specify the width and height of the texture image. Both numbers must be powers of 2, from 8 through 1024.

If *target* is GL_TEXTURE_CUBE_MAP_∗, *width* and *height* must have the same value. Every surface must have the same settings, except for *pixels* (and *target*).

If *pixels* is set to 0 (NULL), a region is allocated but image data is not loaded.

For *internalformat* specify the basic internal format. A GL_INVALID_OPERATION error is generated when *internalformat* and *format* are not the same value. The following table shows how an image's RGBA components correspond to the components in the basic internal format.

**Table 7-4. Correspondence Between RGBA Formats and Basic Internal Formats**

| Basic Internal Format | RGBA | Internal Format |
|---|---|---|
| GL_ALPHA | A | A |
| GL_LUMINANCE | R | L |
| GL_LUMINANCE_ALPHA | R, A | L, A |
| GL_RGB | R, G, B | R, G, B |
| GL_RGBA | R, G, B, A | R, G, B, A |
| GL_HILO8_DMP | R, G | Nx, Ny |

The GL_HILO8_DMP format outputs 0.0 for the B component and 1.0 for the A component.

Unlike in the OpenGL specifications, *level* specifies the number of mipmap levels as a negative value. For example, −2 indicates two mipmap levels, and both −1 and 0 indicate one mipmap level. You cannot separately specify which textures to use for each mipmap level. Instead, set *pixels* to a series of textures, starting with the texture to use as the largest mipmap and ending with the texture to use as the smallest mipmap.

You must set *border* to 0.

**Note:** To configure the GPU access targets and processing to use when allocating the buffer region, pass a bitwise OR of certain specific flag values for target for the

`glTexImage2D()` function. For more information, see 3. Using Data Located in Main Memory in *3DS Programming Manual: Advanced Graphics*.

### 7.3.1. Formats With 4-Bit Components

Texture formats with a *`type`* of `GL_UNSIGNED_BYTE_4_4_DMP` or `GL_UNSIGNED_4BITS_DMP` use 4 bits for a single component, and require a special data ordering.

In a format with a *`type`* of `GL_UNSIGNED_BYTE_4_4_DMP`, two components are stored in a single byte. You can combine this with a *`format`* of `GL_LUMINANCE_ALPHA` or `GL_LUMINANCE_ALPHA_NATIVE_DMP`. The luminance is stored in the most-significant 4 bits and the alpha component is stored in the least-significant 4 bits.

In a format with a *`type`* of `GL_UNSIGNED_4BITS_DMP`, two texels are stored in a single byte. You can combine this with a *`format`* of `GL_LUMINANCE`, `GL_LUMINANCE_NATIVE_DMP`, `GL_ALPHA`, or `GL_ALPHA_NATIVE_DMP`. Viewed as a row of texels, the first component is stored in the least-significant 4 bits, the second component is stored in the most-significant 4 bits, and so on.

**Warning:** If you enable a texture format with a *`type`* of `GL_UNSIGNED_4BITS_DMP` (a 4-bit format) at the same time as another texture format (a non-4-bit format that includes ETC1 compressed textures) and then use them as a multitexture, there are restrictions on the placements of the texture. and then use them as a multitexture, there are restrictions on the placements of the texture.

If you are placing 4-bit textures in VRAM, you must place 4-bit textures and non-4-bit textures in separate memory. In such cases, VRAM-A and VRAM-B are treated as separate memory. Behavior is undefined when textures having different bit formats are placed in the same memory.

There is no restriction on texture arrays when 4-bit textures are placed in main memory.

**Figure 7-1. Bit Layout of Texture Formats That Comprise 4-Bit Components**

## 7.4. Loading Compressed Textures

You can load compressed image data as texture images. **Partial texture loading, with the `glCompressedTexSubImage2D()` function, is not supported.**

**Code 7-4. Definition of the glCompressedTexImage2D Function**

```
void glCompressedTexImage2D(GLenum target, GLint level, GLenum internalformat,
                            GLsizei width, GLsizei height, GLint border,
                            GLsizei imageSize, const void* data);
```

For *target* specify the same value specified in the glTexImage2D() function (Table 7-2). However, shadow textures and gas textures cannot use the regions allocated by this function.

For *width* and *height*, specify the width and height of the texture image. Both numbers must be powers of 2, from 16 through 1024.

The values specified for *level* and *border*, and the restrictions on cube map textures and mipmap textures, are the same as in the glTexImage2D() function. For more information, see 7.3. Loading Texture Images.

The hardware supports only one compressed texture format: ETC1 (Ericsson Texture Compression). You can set *internalformat* to either GL_ETC1_RGB8_NATIVE_DMP or GL_ETC1_ALPHA_RGB8_A4_NATIVE_DMP.

The ETC1 format takes blocks of 4×4 texels in the 24-bit RGB format and compresses them each into 64 bits. An alpha channel is not supported by GL_ETC1_RGB8_NATIVE_DMP, but it is supported by GL_ETC1_ALPHA_RGB8_A4_NATIVE_DMP, which adds 4 bits of alpha component data for each of the 16 texels.

A value of 1 is output for the alpha component when the texture combiner references a compressed texture with a format that does not include an alpha channel.

For *imageSize* specify the number of bytes in the image data. If the original texture image has a width of w and a height of h, *imageSize* can be found using the following equation. The value of *blockSize* is either 8 when there is no alpha channel or 16 when there is.

*imageSize* = (*w* / 4) * (*h* / 4) * blockSize

The ETC1 format handled by the 3DS system is different from the standard OpenGL specifications. 7.10. Native PICA Format explains the differences between this format and the standard specifications. For more information about formats, see the *3DS Programming Manual: Advanced Graphics*.

**Note:** To configure the GPU access targets and processing to use when allocating the buffer region, pass a bitwise OR of certain specific flag values for target for the glCompressedTexImage2D() function. For more information, see 3. Using Data Located in Main Memory in *3DS Programming Manual: Advanced Graphics*.

# 7.5. Copying From the Framebuffer

You can get (copy) an image of the color buffer and depth buffer, which are bound to a framebuffer object as a texture.

## 7.5.1. Copying From the Color Buffer

You can get (copy) an image of the color buffer as a texture.

**Code 7-5. Definition of the glCopyTexImage2D Function**

```
void glCopyTexImage2D(GLenum target, GLint level, GLenum internalformat,
```

```
                        GLint x, GLint y, GLsizei width, GLsizei height,
                        GLint border);
```

For *target* specify the same value specified in the glTexImage2D() function (Table 7-2). All other argument values are also the same, except for the following differences.

- For *internalformat* specify either GL_RGB or GL_RGBA, but data cannot be converted from the color buffer format during the copy (only formats with the same pixel sizes are allowed).
- The values of *x* and *y* specify the starting point of the data region to copy from the color buffer (with the origin at the lower-left corner and the positive axes pointing up and right). The values of *width* and *height* specify the width and height of the region to copy. The values of *x* and *y* must be multiples of 8.
- Only 0 can be specified for *level*.

**Note:** To configure the GPU access targets and processing to use when allocating the buffer region, pass a bitwise OR of certain specific flag values for target for the glCopyTexImage2D() function. For more information, see 3. Using Data Located in Main Memory in *3DS Programming Manual: Advanced Graphics*.

## 7.5.2. Copying From the Color Buffer

You can also copy a partial texture image region from the color buffer.

**Code 7-6. Definition of the glCopyTexSubImage2D Function**

```
void glCopyTexSubImage2D(GLenum target, GLint level,
                         GLint xoffset, GLint yoffset, GLint x, GLint y,
                         GLsizei width, GLsizei height);
```

Data must be copied to a texture image region that was allocated in advance by the glTexImage2D() function.

This function is the same as glCopyTexImage2D except that *xoffset* and *yoffset* specify the coordinates of the region to be copied to (where the origin is the lower-left corner and the positive axes point up and toward the right), and *width* and *height* specify a multiple of 8 (not necessarily a power of two). For more information, see 7.5.1. Copying From the Color Buffer.

## 7.5.3. Copying From the Depth Buffer

If you call the glEnable() function with GL_DEPTH_STENCIL_COPY_DMP passed as an argument, the glCopyTexImage2D() and glCopyTexSubImage2D() functions copy the content of the depth (and stencil) buffer, rather than the color buffer, to a texture as long as depth/stencil copy operations are enabled.

The format of the current depth buffer determines the format of the texture to specify as the copy target. Because the format is not converted during the copy operation, a GL_INVALID_OPERATION error occurs if you attempt to copy data to a texture with an unsupported format. The copied data is identical for both native and non-native texture formats.

**Table 7-5. Depth Buffer Formats and the Corresponding Texture Format and Type**

| | | | |
|---|---|---|---|

| Depth Buffer Format | Texture Format | Texture Type | Components |
|---|---|---|---|
| GL_DEPTH24_STENCIL8_EXT | GL_RGBA<br>GL_RGBA_NATIVE_DMP | GL_UNSIGNED_BYTE | R: Stencil<br>G: D [23:16]<br>B: D [15:8]<br>A: D [7:0] |
| GL_DEPTH_COMPONENT24_OES | GL_RGB<br>GL_RGB_NATIVE_DMP | GL_UNSIGNED_BYTE | R: D [23:16]<br>G: D [15:8]<br>B: D [7:0] |
| GL_DEPTH_COMPONENT16 | GL_HILO8_DMP<br>GL_HILO8_DMP_NATIVE_DMP | GL_UNSIGNED_BYTE | R: D [15:8]<br>G: D [7:0] |

**Note:** The **Components** column shows which bits of the depth value are set in each component. (For example, **Depth [15:8]** indicates that bits 8 through 15 of the depth value are set for that component.)

You can call the `glEnable`, `glDisable`, and `glIsEnabled()` functions and pass `GL_DEPTH_STENCIL_COPY_DMP` as an argument to respectively enable, disable, and determine the current status of depth and stencil copy operations.

**Note:** You cannot use the `glCopyTexImage2D()` and `glCopyTexSubImage2D()` functions in block 32 mode. No errors are generated, but images are not transferred properly. For information about block mode settings, see the *3DS Programming Manual: Advanced Graphics*.

## 7.6. Specifying a Texture as the Render Target

To write rendering results directly to a texture ("render to texture"), take a texture image allocated by `glTexImage2D()` and bind it to a framebuffer object using the `glFramebufferTexture2D()` function. Because rendering results are written directly to each of the special textures used by shadows and gases, you do not need to use this function to specify the special textures as rendering targets.

**Code 7-7. Definition of the glFramebufferTexture2D Function**

```
void glFramebufferTexture2D(GLenum target, GLenum attachment, GLenum textarget,
                            GLuint texture, GLint level);
```

For *`target`*, you can only specify `GL_FRAMEBUFFER`.

For *`attachment`*, specify the data to write to the texture. Specify `GL_COLOR_ATTACHMENT0` if a color buffer and `GL_DEPTH_ATTACHMENT` if a depth buffer.

For *`textarget`* specify the same value specified in the `glTexImage2D()` function (Table 7-2).

For *`texture`*, specify the texture object to bind to the framebuffer object.

Only `0` can be specified for *`level`*.

When a texture is specified as the render target for the depth (and stencil) buffer, the texture format to specify for each depth buffer format is set. The corresponding format is the same as that indicating in 7.5.3. Copying From the Depth Buffer (Table 7-5). When the format of the depth buffer is `GL_DEPTH24_STENCIL8_EXT`, you can set `GL_DEPTH_STENCIL_ATTACHMENT` to *`attachment`*, but it is the same as setting `GL_DEPTH_ATTACHMENT`.

# 7.7. Loading Lookup Tables

The `glTexImage1D()` function loads one-dimensional textures in OpenGL, but it is used to load lookup tables on the 3DS system. A lookup table is a one-dimensional table accessed by procedural textures, fragment lighting, fog, and gases. It cannot be used as a texture.

**Code 7-8. Definition of the glTexImage1D Function**

```
void glTexImage1D(GLenum target, GLint level, GLint internalformat,
                  GLsizei width, GLint border, GLenum format, GLenum type,
                  const GLvoid *pixels);
```

For *target*, specify GL_LUT_TEXTURE*i*_DMP to specify the lookup table to load. The lookup table number is specified in *i* as a number between 0 and one less than the value obtained by passing GL_MAX_LUT_TEXTURES_DMP to *pname* in the `glGetIntegerv()` function. In other words, you can specify a lookup table number in the range from 0 through 31. GL_LUT_TEXTURE*i*_DMP is defined as GL_LUT_TEXTURE0_DMP + *i*.

You can only specify 0 for *level*, GL_FLOAT for *type*, and GL_LUMINANCEF_DMP for *format* and *internalformat*. A GL_INVALID_VALUE error is generated if *level* is nonzero. A GL_INVALID_ENUM error is generated if *type*, *format*, or *internalformat* is not one of these values.

For *width*, specify the number of table elements, and for *pixels*, specify the table elements. The maximum value that can be specified for *width* is 512, which is the same value obtained from the `glGetIntegerv()` function when GL_MAX_LUT_ENTRIES_DMP is passed into *pname*. However, the various reserved fragment processes have unique restrictions on the number of table elements and on the table elements.

To get the ID of the texture object bound to GL_LUT_TEXTURE*i*_DMP, call `glGetIntegerv()` and specify GL_TEXTURE_BINDING_LUT*i*_DMP for the *pname* parameter (where *i* is a value from 0 through 31).

Use the `glTexSubImage1D()` function to rewrite a subset of the lookup table data that has been loaded.

**Code 7-9. Definition of the glTexSubImage1D Function**

```
void glTexSubImage1D(GLenum target, GLint level, GLint xoffset,
                     GLsizei width, GLenum format, GLenum type,
                     const GLvoid *pixels);
```

Data must be copied to a lookup table region that was allocated in advance by the `glTexImage1D()` function.

This function is identical to `glTexImage1D` except for *xoffset* and *width*, which specify the starting element number and the number of elements, respectively. A GL_INVALID_VALUE error is generated if the sum of *width* and *xoffset* exceeds the number of table elements.

The `glCopyTexImage1D()` and `glCopyTexSubImage1D()` functions have not been implemented. Texture parameter settings are also unsupported.

# 7.8. Destroying Texture Objects

You can use the `glDeleteTextures()` function to destroy texture objects that are no longer necessary.

**Code 7-10. Definition of the glDeleteTextures Function**

```
void glDeleteTextures(GLsizei n, const GLuint* textures);
```

# 7.9. Texture Collections

Texture collections are new to the Nintendo 3DS. You can bind texture objects to textures and record them. Using texture collections, you can bind a group of texture objects to a recorded texture type in one operation. Texture collections share their namespace with texture objects and can be created, specified, and destroyed using `glGenTextures`, `glBindTexture`, and `glDeleteTextures`, respectively.

## 7.9.1. Creating Texture Collections

A texture collection operates as a special texture object. Consequently, use the `glGenTextures()` function to generate objects that can be used as texture collections, just as you would for texture objects.

## 7.9.2. Specifying Texture Collections

Use the `glBindTexture()` function to specify a texture object to use as a texture collection. Call this function, and for the *`target`* parameter, specify `GL_TEXTURE_COLLECTION_DMP`. By default, the object with a name of 0 (with a value of 0 passed into *`texture`*) is a texture collection.

After this function is called, the texture collection records the texture objects bound by `glBindTexture` to each texture type (2D texture, cube map texture, and lookup table). The function overwrites any other existing binding to the same texture. Settings recorded in the texture collection remain until the collection is switched.

When the texture collection is switched, all texture objects recorded in the new texture collection are bound as the new textures.

## 7.9.3. Destroying Texture Collections

Texture collections can be destroyed with the `glDeleteTextures()` function, just like texture objects. Even if a texture collection is destroyed, it does not affect texture bindings to the texture objects recorded in that vertex state collection.

The `glDeleteTextures()` function does not immediately destroy a texture collection that is in use. A texture collection remains in use until it is switched with another one. You cannot destroy the default texture collection. Calls to `glDeleteTextures` on the default texture collection are ignored.

# 7.10. Native PICA Format

The GPU's texture unit supports a different texture format than the OpenGL specifications. The format that is actually supported by the texture unit is called the native PICA format. Because libraries do not convert textures loaded in native PICA format, it is more efficient than the standard format.

There are three major differences between this format and the OpenGL specifications.

- Byte order
  Bytes are written in a different order because of the way internal addresses are processed.

- V-Flip
  The two formats use v-axes that run in opposite directions, when placing texels using UV coordinates.

- Addressing
  Texels and compressed data blocks are written in different orders, because of differences between linear addressing (OpenGL) and block addressing (native PICA format).

In the CTR system, v-flip, addressing, and byte-order conversions are run, in that order, on an uncompressed texture (loaded by `glTexImage2D`), to convert it from the OpenGL format into the native PICA format. The system converts a compressed texture (loaded by `glCompressedTexImage2D`) by running V-flip, ETC compression, addressing, and finally, byte-order conversions. Note that v-flip conversion must precede ETC compression.

## 7.10.1. Byte Order Swapping

When the format uses more than one byte to represent a single texel and the texture was defined by calling `glTexImage2D()` and passing `GL_UNSIGNED_BYTE` for *`type`*, the number of bytes swapped (replaced) is the same as the number of bytes that represent a texel.

Data in a compressed format is swapped one block (8 bytes for 4×4 texels) at a time. However, if the format has an alpha channel (the first 8 bytes), the alpha portion is not byte-swapped.

**Figure 7-2. Four-Byte Swap**

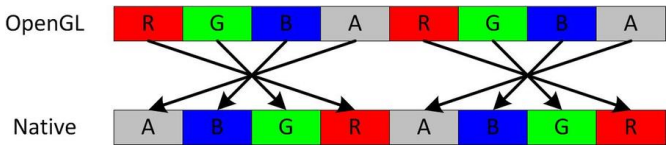



**Figure 7-3. Three-Byte Swap**

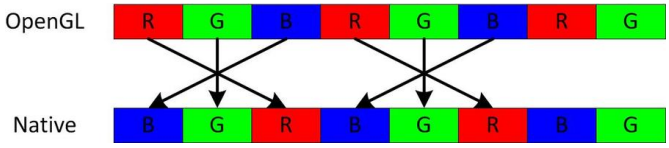



**Figure 7-4. Two-Byte Swap**

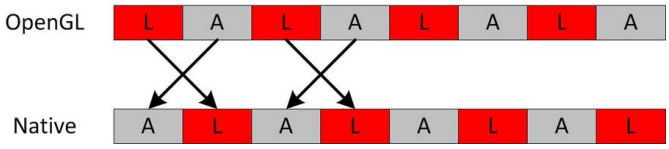



**Figure 7-5. Byte Swap in a Compressed Format (No Alpha Channel)**

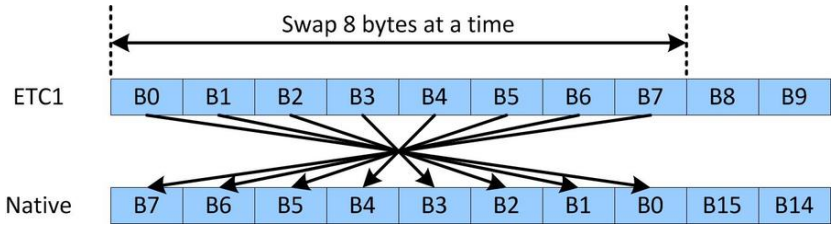
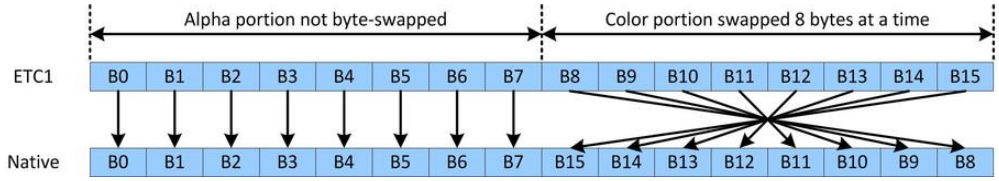



**Figure 7-6. Byte Swap in a Compressed Format (With Alpha Channel)**




## 7.10.2. V-Flip Differences

In the OpenGL specifications, images are converted into data starting with the texel at UV coordinates (0.0, 0.0). In the native PICA format, images are converted into data, starting with the texel at UV coordinates (0.0, 1.0). This is the same for both uncompressed and compressed textures.

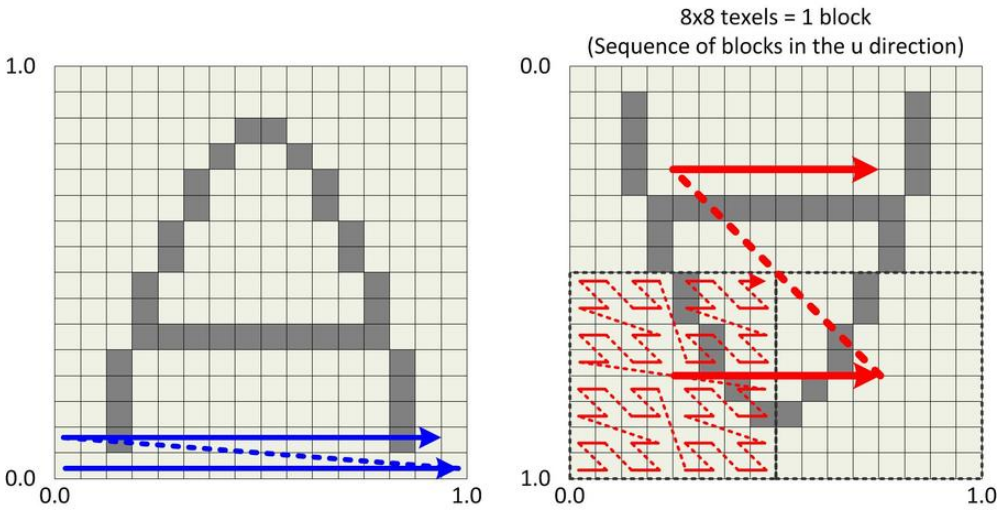
## 7.10.3. Differences in Addressing

In the OpenGL specifications, texels are stored consecutively in the U direction (linear addressing). In the native PICA format, blocks of texels are stored, consecutively, in the U direction, but texels within those blocks are stored in a zigzag pattern (block addressing).

### 7.10.3.1. Uncompressed Textures

Texels are stored in a zigzag pattern within each block of 8×8 texels (as shown by the red lines in the figure), but the blocks are stored consecutively in the U direction.

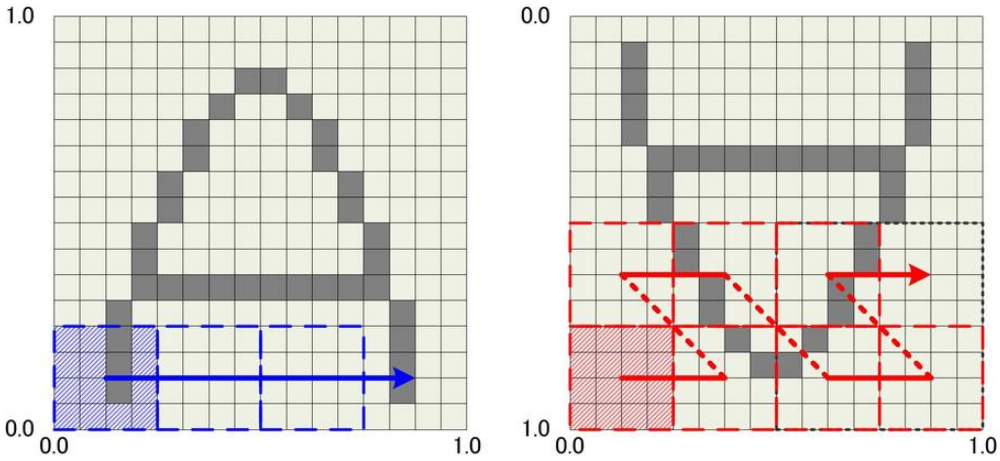
**Figure 7-7. Texel Order in an Uncompressed Texture**




### 7.10.3.2. Compressed Textures

A compressed texture compresses each block of 4×4 texels. In the OpenGL specifications, blocks are stored consecutively in the U direction (shown by the blue line in the figure). In the native PICA format, blocks are stored in a zigzag pattern in meta-blocks of 2×2 blocks. The meta-blocks themselves are stored consecutively in the U direction (shown by the red line in the figure).

**Figure 7-8. Texel Order in a Compressed Texture**




For more information about compressed textures, see the *3DS Programming Manual: Advanced Graphics*.



# 8. Vertex Shaders

A vertex shader converts coordinates, applies shadows, and performs other operations on vertex attributes that are input. The 3DS system has four vertex processors, each of which can handle vector data comprising four, 24-bit floating-point values. The input vertex data is processed, in parallel, by the

four vertex processors. Although the processors can run general-purpose calculations, they cannot read data from or write data to VRAM.

Like OpenGL ES 2.0, the 3DS system does not make a clear distinction between coordinates, normals, and other attributes stored with vertex data. Attributes determine how a vertex shader processes and outputs its input data. In general, the following types of attributes are input as vertex data for 3D graphics processing.

- Vertex coordinates
- Normal vectors
- Tangent vectors
- Texture coordinates
- Vertex color

Vertex shaders are written in an assembly language that is unique to the PICA graphics core. We recommend that you refer to the Vertex Shader Reference as you continue to read this chapter.

## 8.1. Input Vertex Data

Vertex data input by the application is passed to the vertex shader through input registers that are bound to vertex attribute numbers. Using `#pragma bind_symbol`, the vertex shader specifies the names and registers for input data.

**Code 8-1. Binding Data Names and Registers (in Shader Assembly)**

```
#pragma bind_symbol(AttribPosition.xyzw, v0, v0)
```

In this code sample, the `xyzw` component for the `v0` input register is bound to the `AttribPosition` data name. You cannot bind more than one data name to the same input register. The second and third arguments must also take the same value (or the third argument must be omitted).

The application uses the `glBindAttribLocation()` function to bind vertex attribute numbers and data names, the `glEnableVertexAttribArray()` function to enable bound vertex attribute numbers, and `glVertexAttribPointer` or another function to input vertex data.

**Code 8-2. Binding Vertex Attribute Numbers and Inputting Vertex Data**

```
glBindAttribLocation(program, 0, "AttribPosition");
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 0, pointer);
```

In this code sample, the vertex attribute number 0 is bound to data having the name `AttribPosition`, and vertex attributes for four components are specified. **The register number and the vertex attribute number do not have to be the same.**

Input registers that are not bound by `#pragma bind_symbol` have undefined values. Any input vertex data that is not a `GL_FLOAT` value is automatically converted into a `GL_FLOAT` value. However, note that the data is not normalized.

When a vertex buffer is used, the vertex attribute data type and data size combination affects transfer speed of the vertex data. For more information, see 15.13. Effect of Vertex Attribute Combinations on

Vertex Data Transfer Speed.

> **Warning:** If vertex data is input through the `glVertexAttribPointer()` function, the fourth argument (specifying whether to normalize values) is ignored because it is not supported by the hardware. The vertex shader must explicitly normalize values because the specified value has no effect.
>
> You cannot specify `GL_FIXED` or `GL_UNSIGNED_SHORT` for the *`type`* parameter of the `glVertexAttribPointer()` function. If `GL_FLOAT` or `GL_SHORT` is specified, *`ptr`* must be a pointer that is 4-byte aligned or 2-byte aligned, respectively.

While a vertex shader is processing a single vertex, it must load vertex data from at least one input register (even a single component), or it may not run properly.

## 8.2. Vertex Data Output

The vertex shader's processing results are written to the output registers that are mapped to output vertex attributes, and in this way the results are passed on to the next processing stage. Using `#pragma output map`, the vertex shader specifies the vertex attribute names and registers for output data.

**Code 8-3. Mapping Registers to Output Vertex Attributes and Writing to the Registers (in Shader Assembly)**

```
#pragma output_map(position, o0)
mov     o0,     v0
```

In this code sample, vertex coordinates are mapped to the `o0` output register. Data is output as vertex coordinates by writing data to `o0`.

Because only a reserved shader can be used for fragment processing, the vertex shader's output vertex attributes will be fixed in advance. Output vertex attributes have the following names. The components that the vertex shader must set for the output attributes are fixed.

**Table 8-1. Output Vertex Attributes**

| Attribute Name | Output Attribute | Required Components |
|---|---|---|
| `position` | Vertex coordinates. | 4 components: x, y, z, and w. |
| `color` | Vertex color. | 4 components: R, G, B, and A. |
| `texture0` | Texture coordinate 0 | 2 components: u and v. |
| `texture0w` | Texture coordinate 0 | 1 component: w. |
| `texture1` | Texture coordinate 1 | 2 components: u and v. |
| `texture2` | Texture coordinate 2 | 2 components: u and v. |
| `quaternion` | Quaternions. | 4 components: x, y, z, and w. |
| `view` | View vector. | 3 components: x, y, and z. |
| `generic` | General-purpose attribute. | Any number of components that are used by the geometry shader. |

The vertex shader must write some value to every component (`x`, `y`, `z`, and `w`) of registers that have

been mapped. Dummy values must be written to any unmapped (unused) components in the registers specified by `#pragma output_map`.

The vertex shader forces processing to end when values have been written to all mapped registers. It then moves on to process the next vertex (the `end` instruction must be called). In other words, after the last attribute data has been written to a register, later instructions might not be executed.

You can only write to (each component of) a single output register once while processing a single vertex. Correct operation is not guaranteed if data is written to the same component (of the same register) more than once.

Except for the `generic` attribute, output vertex attributes can be mapped to no more than seven output registers. To map eight or more non-`generic` output vertex attributes, you must map multiple attributes to a single register. In this case, multiple vertex attributes (up to a total of four components) must be packed into a single register. For example, you could map `texture1` and `texture2` to `o0.xy` and `o0.zw`, respectively.

## 8.3. Uniform Settings

When `#pragma bind_symbol` binds data names to non-input registers (floating-point constant registers, Boolean registers, or integer registers), the application can use the `glUniform*()` functions to set values in each register. You can transpose matrices within the `glUniformMatrix*()` functions (convert the matrices from column-major to row-major order), by specifying `GL_TRUE` for *`transpose`*.

Sample code follows for both the vertex shader and the application.

**Code 8-4. Binding Data Names to Non-Input Registers (in Shader Assembly)**

```
#pragma bind_symbol ( ModelViewMatrix , c0 , c3 )
#pragma bind_symbol ( LoopCounter0 , i1 , i1 )
#pragma bind_symbol ( bFirst , b2 , b2 )
#pragma bind_symbol ( Scalar.x , c4, c4 )
```

**Code 8-5. Setting Uniforms**

```
uniform_location = glGetUniformLocation ( program , "ModelViewMatrix" );
GLfloat matrix[4][4];
glUniform4fv ( uniform_location , 4 , matrix );

GLfloat scalar_value;
uniform_location = glGetUniformLocation ( program , "Scalar" );
glUniform1f ( uniform_location , scalar_value );

uniform_location = glGetUniformLocation ( program , "bFirst" );
glUniform1i ( uniform_location , GL_TRUE );

GLint loop_setting[3] = { 4 , 0 , 1 } ; // loop_count - 1 , init , step
uniform_location = glGetUniformLocation ( program , "LoopCounter0" );
glUniform3iv ( uniform_location , loop_setting );
```

The sample code specifies component `x` for the data name `Scalar`. Components can be specified this way when binding a floating-point constant register. You must specify components consecutively, in `xyzw` order. In other words, `xy`, `zw`, or `yzw` can be specified but not `xz`, `yw`, or `xyw`.

Integer registers are used to control the `loop` instruction in a shader program. In a register that is 24 bits wide, the loop count is assigned to bits 0 through 7, the initial value is assigned to bits 8 through 15, and the increment value is assigned to bits 16 through 23. The `loop` instruction initializes the loop counter register to a default value, and then repeatedly executes the instructions between `loop`

and endloop once more than the specified loop count. The loop counter register is incremented only by the increment value each time through the loop.

## 8.4. Notes for the Clip Coordinate System

The vertex shader outputs Z components in a clip coordinate system that differs from the one used in OpenGL ES.

OpenGL ES clips coordinates between –Wc and Wc, but the 3DS system clips them between 0 and –Wc (the sign is reversed). To use projective transformation matrices that are compatible with OpenGL ES, applications must convert the range from –Wc to Wc into the range from 0 to –Wc.

### Converting Projective Transformation Matrices in the Application

Make the following conversion and set the resulting projection matrix as a uniform.

**Code 8-6. Converting to an OpenGL ES-Compatible Projective Transformation Matrix**

```
GLfloat projection[16];
projection[2] = (projection[2] + projection[3]) * (-0.5f);
projection[6] = (projection[6] + projection[7]) * (-0.5f);
projection[10] = (projection[10] + projection[11]) * (-0.5f);
projection[14] = (projection[14] + projection[15]) * (-0.5f);
```

### Converting in a Vertex Shader

Apply the projection conversion as follows.

**Code 8-7. Converting to an OpenGL ES-Compatible Projective Transformation Matrix (in Shader Assembly)**

```
#pragma output_map(position, o0)
#pragma bind_symbol(attrib_position, v0)
#pragma bind_symbol(modelview, c0, c3)
#pragma bind_symbol(projection, c4, c7)
def     c8, -0.5, -0.5, -0.5, -0.5

// Model View Transformation
dp4     r0.x, v0, c0
dp4     r0.y, v0, c1
dp4     r0.z, v0, c2
dp4     r0.w, v0, c3
// Projective Transformation
dp4     o0.x, r0, c4
dp4     o0.y, r0, c5
mov     r1, c6
add     r1, r1, c7
mul     r1, r1, c8
dp4     o0.z, r0, r1
dp4     o0.w, r0, c7
```

## 8.5. Vertex Cache

Some of the vertex data created or processed by the vertex shader is saved in a cache. The vertex

shader does not process input vertex data that is determined to be the same as the original vertex data saved in the cache, based on its vertex indices. Instead, the processed data in the cache is sent to the next process. The same vertex data is often processed more than once when it is input using `GL_TRIANGLES`, but this can be avoided if there is processed vertex data in the cache already.

The following conditions must be satisfied to use the vertex cache.

- Vertex data must be input in a format that accesses vertex indices. In short, `glDrawElements` must be called to input vertex data.
- Input vertex data must use a vertex buffer.

The vertex cache can save 32 vertex data entries. It is implemented with a proprietary algorithm that resembles the functionality of the LRU (least recently used) algorithm.

When the repeatedly accessed vertex data contains no more than 32 vertices, there is a higher chance of a cache hit. But the efficiency of the vertex cache is affected by conditions other than index order, including the usage state of the memory holding the index array, and the length of the shader executing as the vertex shader. For these reasons, the optimal index depends on the content, and there may not be a definitive answer.

## 8.6. Querying the Vertex Shader

You can query the vertex shader for information about active vertex attributes and uniforms.

### 8.6.1. Getting Vertex Attribute Information

You can use the `glGetActiveAttrib()` function to get attribute information for the vertex data input to the vertex shader.

**Code 8-8. Definition of the glGetActiveAttrib Function**

```
void glGetActiveAttrib(GLuint program, GLuint index, GLsizei bufsize,
                       GLsizei* length, GLint* size, GLenum* type, char* name);
```

A `GL_INVALID_OPERATION` error is generated if *`program`* specifies an unlinked or otherwise invalid program object.

For *`index`*, specify a value between `0` and one less than the number of vertex attributes obtained from the `glGetProgramiv()` function, when it is called on the program object specified by *`program`*, when `GL_ACTIVE_ATTRIBUTES` is specified for *`pname`*. A `GL_INVALID_VALUE` error is generated if the specified value is negative or greater than or equal to the number of vertex attributes.

For *`bufsize`*, specify the size of the array specified in *`name`*. A `GL_INVALID_VALUE` error is generated if a negative value is specified.

The vertex attribute's type is returned in *`type`*. The vertex attribute's size is returned in *`size`*. This is the number of values indicated by *`type`* that are required to represent the vertex attribute.

The vertex attribute's name is returned in *`name`*. If there are more than *`bufsize`* characters in the vertex attribute's name, up to *`bufsize`* - 1 characters are stored with a terminating character (`NULL`) added at the end. The number of characters in *`name`* is returned in *`length`* (excluding the

terminating null character).

## 8.6.2. Getting Uniform Information

You can use the `glGetActiveUniform()` function to get uniform information registered with a program object. This is not limited to the vertex shader; you can also get information about uniforms used by the geometry shader and reserved fragment shader, which are described later.

**Code 8-9. Definition of the glGetActiveUniform Function**

```
void glGetActiveUniform(GLuint program, GLuint index, GLsizei bufsize,
                        GLsizei* length, GLint* size, GLenum* type, char* name);
```

A `GL_INVALID_OPERATION` error is generated if *`program`* specifies an unlinked or otherwise invalid program object.

For *`index`*, specify a value between `0` and one less than the uniform information count obtained from the `glGetProgramiv()` function when it is called on the program object specified by *`program`* and when *`pname`* is `GL_ACTIVE_UNIFORMS`. A `GL_INVALID_VALUE` error is generated if the specified value is negative or greater than or equal to the uniform information count.

For *`bufsize`*, specify the size of the array specified in *`name`*. A `GL_INVALID_VALUE` error is generated if a negative value is specified.

The type of value for the uniform setting is returned in *`type`*. The value's size is returned in *`size`*. This is the number of elements indicated by *`type`* that are required to represent the uniform setting. For example, `GL_FLOAT_VEC4` is stored in *`type`* and 4 is stored in *`size`* for a 4×4 matrix, such as the modelview matrix.

The uniform's name is returned in *`name`*. If there are more than *`bufsize`* characters in the uniform's name, up to *`bufsize`* − 1 characters are returned with a terminating character (`NULL`) added at the end. The number of characters in *`name`* is returned in *`length`* (excluding the terminating null character).

## 8.6.3. Setting Categories

The `glGetActiveAttrib()` and `glGetActiveUniform()` functions can get the following types of values.

**Table 8-2. List of Setting Categories**

| Type | Type | Number of Components | Major Use |
|---|---|---|---|
| `GL_FLOAT` | `float` | 1 | Bias and scale values. |
| `GL_FLOAT_VEC2` | `float` | 2 | Viewport settings. |
| `GL_FLOAT_VEC3` | `float` | 3 | Colors (RGB) and directional vectors. |
| `GL_FLOAT_VEC4` | `float` | 4 | Colors (RGBA) and transformation matrices. |
| `GL_INT` | `int` | 1 | Mode settings. |
| `GL_INT_VEC3` | `int` | 3 | Combiner source input. |
| `GL_BOOL` | `bool` | 1 | Enabling and disabling features. |
| `GL_SAMPLER_1D` | `int` | 1 | Specifying lookup tables. |

Lists categories that are actually used by uniforms in each shader.

## 8.7. Getting and Setting the Values of Multiple Uniforms

Functions are provided for getting and setting multiple uniforms, concurrently, on the 3DS system.

By calling the `glUniformsDMP()` function, you can concurrently set values in multiple uniforms for the program object that is currently bound.

**Code 8-10. Setting a Group of Uniforms**

```
void glUniformsDMP(GLuint n, GLint* locations, GLsizei* counts,
                   const GLuint* value);
```

For *n*, specify the number of uniforms to set.

For *locations*, specify a pointer to an array storing *n* uniform locations (which can be obtained by `glGetUniformLocation`). For *counts*, specify a pointer to an array storing the number of elements in the *n* uniforms. *count* for the `glUniform*()` functions corresponds to the number of uniform elements. Fill the array specified by *counts* with the number of elements to set for each array uniform and 1 for each non-array uniform.

For *value*, specify a pointer to an array storing the values to set in the uniforms. Because each uniform has a different amount of data, the indices for the values to store in *value* are not necessarily the same as the indices in *locations* and *counts* for the corresponding uniforms. You can mix both `GLfloat` and `GLuint` data in the uniforms that you set. Store 32-bit `GLfloat` data for the values to set in `GLfloat` uniforms.

This function does not perform any error-checking. Behavior is undefined if you specify an invalid value for any argument.

By calling the `glGetUniformsDMP()` function, you can concurrently get values from multiple uniforms for a specified program object.

**Code 8-11. Getting a Group of Uniforms**

```
void glGetUniformsDMP(GLuint program, GLuint n, GLint* locations,
                      GLsizei* counts, GLuint* params);
```

For *program*, specify the program object for which to get uniform values.

For *n*, specify the number of uniforms to get.

For *locations*, specify a pointer to an array storing *n* uniform locations (which can be obtained by `glGetUniformLocation`). For *counts*, specify a pointer to an array storing the number of elements in the *n* uniforms. The number of uniform elements is either 1 (for a non-array uniform) or the number of array elements from which to get values for an array uniform.

For *params*, specify a pointer to an array used to get the uniform values. Because each uniform has a different amount of data, the indices for the values stored in *params* are not necessarily the same as the indices in *locations* and *counts* for the corresponding uniforms. Both `GLfloat` and `GLuint` data can be mixed in the uniforms from which values are obtained. The `GLfloat` values obtained from `GLfloat` uniforms are stored as 32-bit data in *params*.

This function does not perform any error-checking. Behavior is undefined if you specify an invalid value for any argument.

## 8.8. Other Notes and Cautions

You can access uniform values with the `glUniform*()` and `glGetUniform*()` functions, using uniform locations that can be obtained with the `glGetUniformLocation()` function. By adding an offset to a location, you can specify and access a specific element in an array uniform. For example, the second element of an array uniform is accessed when 1 is added to the uniform's location.

A uniform's location is fixed when the `glLinkProgram()` function is called and the value originally differs for each program object. The `glUniform*()` functions generate an error if the location is not related to the current program object. The `glGetUniform*()` functions generate an error if the location is related to a program object other than *`program`*. However, the `glUniform*` and `glGetUniform*()` functions do not generate errors if the programs being queried are reserved fragment shaders and the location is specified using a bitwise OR with `0xFFF80000`.

Do not include vertex attributes in `#pragma output_map` definitions when they do not need to be output. If you do not follow this advice, useless instructions will be required because values must be written to all defined vertex attributes (output registers) when the vertex shader is run. Some output attributes also involve clock instructions for parts of the GPU circuit, and their output can needlessly run down the battery.

The maximum number of vertex attributes when rendering without using a vertex buffer is 16, and the maximum number of vertex attributes that can use a vertex buffer is 12. However, if 12 vertex attributes are rendered using a vertex buffer, you must take care about the limitations in vertex data placement (see 6.8.1. Restrictions Affecting Only glDrawElements).

In the vertex shader assembler, you can define up to 16 vertex attributes, but if the maximum number as determined according to the conditions above is exceeded, a `GL_INVALID_OPERATION` error may be generated when the rendering function is called.



# 9. Geometry Shaders

Geometry shaders use vertex shader output to generate primitives and output an arbitrary number of vertices.

In OpenGL ES 2.0, you can specify `GL_POINTS`, `GL_LINES`, `GL_LINE_STRIP`, or `GL_LINE_LOOP` to the `glDrawElements` or `glDrawArrays()` function to render primitives, but on the 3DS system, you must use the geometry shader to generate non-triangle primitives. Triangles are rendered with `GL_TRIANGLES`, `GL_TRIANGLE_STRIP`, and `GL_TRIANGLE_FAN` because they are included in OpenGL ES 2.0, but multisample rendering is not supported.

As described in 5. Shader Programs, the SDK provides geometry shaders for the Nintendo 3DS system. You can use the following geometry shaders.

- Point shaders
- Line shaders
- Silhouette shaders
- Catmull-Clark subdivision shaders
- Loop subdivision shader
- Particle system shader

These geometry shaders cannot be used alone. Always load a binary that also has a linked vertex

shader, and also make sure that you use the vertex shader. When a geometry shader is used, one of the four vertex processors is used as the geometry processor. The 15th Boolean register (b15) is reserved for the geometry shader.

**Figure 9-1. Processor Configurations When the Geometry Shader Is Used and Not Used**

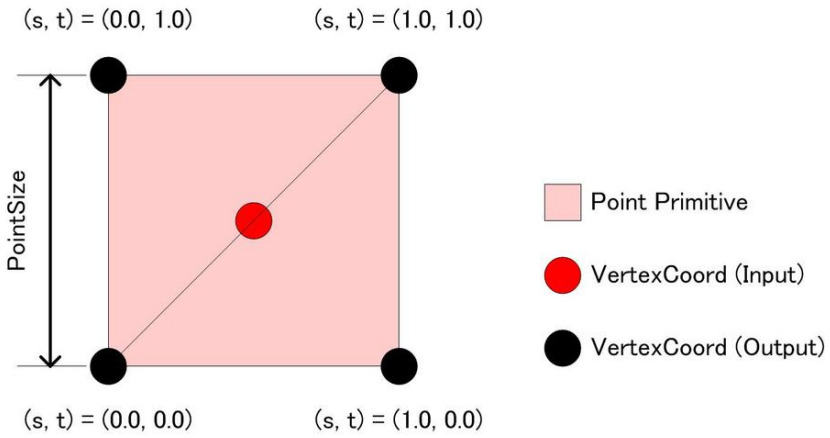
Vertex shader output is used as input data to the geometry shader. Each geometry shader requires specific vertex attributes and a specific input order and can only use some vertex attributes. The vertex shader must output these values correctly. Data is input to the geometry shader in order, starting with the smallest register number output by the vertex shader. Output vertex attributes are defined by `#pragma output_map`, but any `generic` attribute names are only used by the geometry shader, without being handled by some later process (such as fragment processing). Geometry shaders have reserved uniforms.

The point shader, for example, uses reserved uniforms to configure the viewport and other settings. Because all of these reserved uniforms initially have undefined values, they must be set by the application.

## 9.1. Point Shaders

The point shader generates two triangle primitives to draw a square (a point primitive) centered at the specified vertex coordinates, with sides as long as the specified point size. The point sprite shader is a point shader that outputs texture coordinates **s, t** (shown in the following figure), **r** (fixed at 0.0), and **q** (fixed at 1.0), for applying textures to point primitives. Neither shader supports grid adjustments or multisample rendering.

**Figure 9-2. Drawing Point Primitives**



## 9.1.1. Shader Files

The shader file to link with the vertex shader is determined by two factors: the number of vertex attributes output for fragment processing that are also not required by the point shader, and the number of texture coordinates output by the point sprite shader.

Points: DMP_point*N*.obj

Point sprites: DMP_pointSprite*N*_*T*.obj

Where *N* is the number of vertex attributes that are not required by the point (or point sprite) shader, and *T* is the number of texture coordinates.

## 9.1.2. Reserved Uniform

There are reserved uniforms for configuring the viewport, and for enabling or disabling distance attenuation on the point size. These reserved uniforms must be set by the application because they initially have undefined values.

#### Viewport

Use the `glUniform2fv()` function to set the width and height of the viewport in the reserved uniform `dmp_Point.viewport`.

#### Distance Attenuation

Use the `glUniform1i()` function to set the reserved uniform for distance attenuation on the point size, `dmp_Point.distanceAttenuation`. A value of `GL_TRUE` enables distance attenuation, and `GL_FALSE` disables it. The point size is multiplied by the clip coordinate `Wc` when distance attenuation is disabled. This cancels out the division by `Wc` during the conversion to window coordinates and prevents the size of displayed point primitives from being affected by distance.

**Table 9-1. Reserved Uniforms for Point Shaders**

| Reserved Uniform | Type | Value to Set |
|---|---|---|
| `dmp_Point.viewport` | `vec2` | The viewport using the following equation.<br>(1 / viewport.width, 1 / viewport.height) |
| `dmp_Point.distanceAttenuation` | `bool` | Specifies whether the point size is affected by distance attenuation.<br>Specify `GL_TRUE` to enable distance attenuation, and `GL_FALSE` otherwise. |

## 9.1.3. Vertex Shader Settings

The point shader requires vertex coordinates and a point size to render point primitives. The vertex shader outputs the vertex coordinates and then the point size in order (starting with the smallest output register number). This is followed by the texture coordinates for the point sprite shader. If multiple texture coordinate pairs are used, they must be packed two at a time into the `xy` and `zw` components of a single output register.

Output vertex attributes must be set as follows: `position` for the vertex coordinates, `generic` for the point size, and `texture0` through `texture2` for the texture coordinates.

If the vertex shader outputs the vertex color in addition to the vertex coordinates and point size required by the point shader, link the `DMP_point1.obj` shader file and use the following `#pragma output_map` statements in the vertex shader.

**Code 9-1. Sample Output Register Settings When the Point Shader Is Used (in Shader Assembly)**

```
#pragma output_map ( position , o0 )
#pragma output_map ( generic , o1 )
#pragma output_map ( color , o2 )
```

If the vertex shader outputs the vertex color in addition to the vertex coordinates, point size, and two texture coordinate pairs required by the point sprite shader, link the `DMP_pointSprite1_2.obj` shader file and use the following `#pragma output_map` statements in the vertex shader.

**Code 9-2. Sample Output Register Settings When the Point Sprite Shader Is Used (in Shader Assembly)**

```
#pragma output_map ( position , o0 )
#pragma output_map ( generic , o1 )
#pragma output_map ( texture0 , o2.xy )
#pragma output_map ( texture1 , o2.zw )
#pragma output_map ( color , o3 )
```

The point sprite shader outputs texture coordinates for point sprites that replace the input texture coordinates. Note that the vertex shader must write dummy values to the output registers for the texture coordinates that are replaced by the point sprite shader.

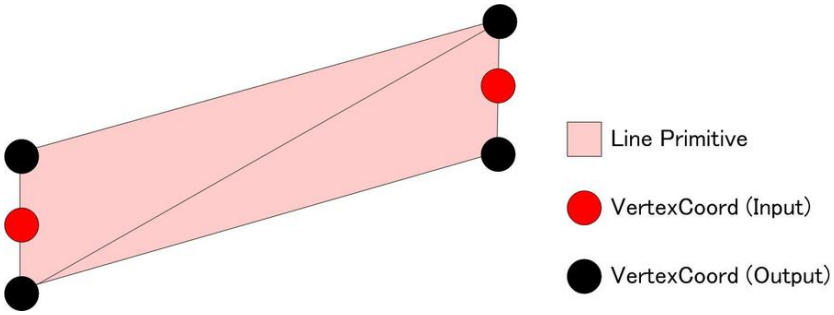
### 9.1.4. Input Vertex Data

To generate primitives with the point shader, call the `glDrawElements` or `glDrawArrays()` function and specify `GL_GEOMETRY_PRIMITIVE_DMP` for *`mode`*.

## 9.2. Line Shaders

A line shader generates two triangle primitives to draw lines (a line primitive) that connect two points specified by vertex coordinates. You can use a reserved uniform to set the line width. The line shader does not support grid adjustments or multisample rendering.

**Figure 9-3. Drawing Line Primitives**




Coordinates are generated for the four vertices that make a rectangle (parallelogram) from the slope

and width of the line segment connecting the specified vertex coordinates. The coordinates of the four vertices are generated from the input vertex coordinates in the Y direction (or the X direction for some line segment slopes).

## 9.2.1. Shader Files

The shader file to link with the vertex shader is determined by the number of vertex attributes output for fragment processing that are also not required by the line shader.

Vertex coordinates can be specified for separate lines or line strips. Separate lines are drawn using two sets of vertex coordinates per line. The coordinates of the first two vertices are the same for both a line strip and separate lines, but the next separate line is drawn from the coordinates of the second and third vertices. In other words, starting with the third vertex, each vertex is used together with the previous one to draw a single, connected line.

Separator lines: DMP_separateLine*N*.obj

Strip lines: DMP_stripLine*N*.obj

Where *N* is the number of vertex attributes not required by the line shader.

## 9.2.2. Reserved Uniform

There is a reserved uniform for setting the line width. The reserved uniform for the line width must be set by the application because its initial value is undefined.

### Line Width

Use the `glUniform4fv()` function to set the reserved uniform for the line width, `dmp_Line.width`, using values calculated from both the line width and the width and height of the viewport.

**Table 9-2. Reserved Uniforms for the Line Shader**

| Reserved Uniform | Type | Value to Set |
|---|---|---|
| `dmp_Line.width` | `vec4` | The following expression calculates the line width.<br>(viewport.width / line.width,<br>viewport.height / line.width,<br>viewport.width * viewport.height,<br>2 / line.width) |

## 9.2.3. Vertex Shader Settings

The line shader requires vertex coordinates to render line primitives. The vertex shader outputs the vertex coordinates, starting with the smallest output register number.

One output vertex attribute must be set: the `position` attribute of the vertex coordinates.

If the vertex shader outputs the vertex color in addition to the vertex coordinates required by the separate line shader, link the `DMP_separateLine1.obj` shader file and use the following `#pragma output_map` statements in the vertex shader.

**Code 9-3. Sample Output Register Settings When a Line Shader Is Used (in Shader Assembly)**

```
#pragma output_map ( position , o0 )
#pragma output_map ( color , o1 )
```

The line strip shader works the same way as the separate line shader. Its shader file is `DMP_stripLine1.obj`.

### 9.2.4. Input Vertex Data

To generate primitives with the line shader, call the `glDrawElements()` or `glDrawArrays()` function and pass `GL_GEOMETRY_PRIMITIVE_DMP` for the `mode` parameter.

## 9.3. Silhouette Shaders

Silhouette shaders generate and render silhouettes around object edges. You can use silhouette edges to render object contours and, when combined with the shadow feature, soft shadows.

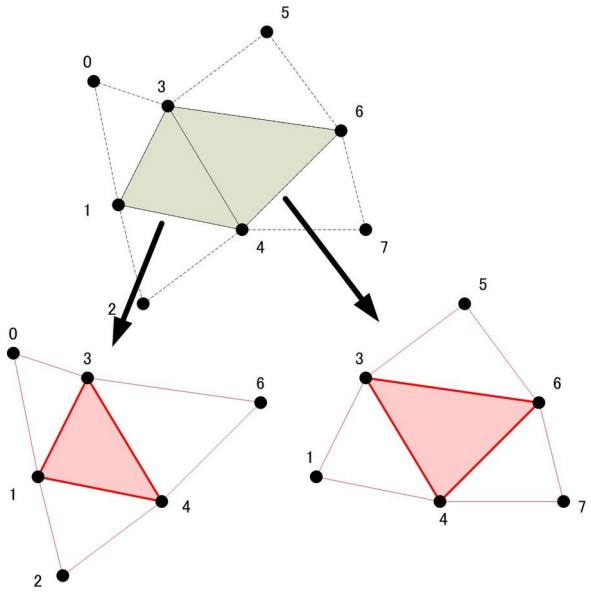
To generate silhouette edges, the silhouette shader needs a primitive called a *triangle with neighborhood*, or TWN for short.

### 9.3.1. Triangle With Neighborhood

This section describes one of the triangles that make up the objects used to render silhouette edges.

This is called a center triangle. A *triangle with neighborhood* (TWN) comprises a center triangle and the three (adjacent) triangles that share an edge with it.

**Figure 9-4. Sample Triangles With Neighborhood**



One TWN comprises the center triangle with vertices 3, 1, and 4, in addition to the three triangles defined by vertices 0, 1, and 3; 2, 4, and 1; and 6, 3, and 4. When vertices 3, 4, and 6 form the center triangle, vertices 3, 1, and 4 represent a triangle in a separate TWN.

TWNs are used to detect the silhouette edges of center triangles. You can make an object from TWNs to render silhouette edges on it.

## 9.3.2. Shader Files

TWN vertices can be input to a silhouette shader in two ways: (1) either as silhouette triangles, one TWN at a time (just like normal triangles), or (2) as continuous silhouette strips of adjoining TWNs.

Silhouette triangles: DMP_silhouetteTriangle.obj

Silhouette strips: DMP_silhouetteStrip.obj

## 9.3.3. Reserved Uniform

Silhouette shaders have the following reserved uniforms. These reserved uniforms must be set by the application because they initially have undefined values.

#### Polygon Facing

Use the `glUniform1i()` function to configure how the silhouette shader determines whether a polygon is front-facing (`dmp_Silhouette.frontFaceCCW`). Specify `GL_TRUE` or `GL_FALSE` if `GL_CCW` or `GL_CW`, respectively, have been passed to the `glFrontFace()` function for object vertex input.

#### Silhouette Edge Width

Use the `glUniform2fv()` function to set the silhouette edge width (`dmp_Silhouette.width`) to a value calculated by multiplying the normal vector's x direction by a coefficient.

You can configure the effect of a vertex's w component (`dmp_Silhouette.scaleByW`). Multiplying this component by the silhouette edge width disables distance attenuation. Call the `glUniform1i()` function and specify a value of `GL_TRUE` to multiply by the w component, or specify `GL_FALSE` to fix the silhouette edge's w component at 1.0.

#### Silhouette Edge Colors

Use the `glUniform4fv()` function to set the color of the silhouette edge (`dmp_Silhouette.color`) with R, G, B, and A values.

#### Open Edges

An open edge is an edge of the center triangle that is not shared with any other triangles. You can configure open edges (`dmp_Silhouette.acceptEmptyTriangles`) to always be drawn (`GL_TRUE`) or not (`GL_FALSE`).

Open edges differ from silhouette edges in that they are drawn like line primitives, without using vertex normals. As a result, they may look different from silhouette edges at some angles. Some settings are specific to open edges.

### Open Edge Width

Use the `glUniform4fv()` function to set the open edge width (`dmp_Silhouette.openEdgeWidth`) using values calculated from a specified width and the viewport's width and height (just like a line shader).

### Open Edge Colors

Use the `glUniform4fv()` function to set the open edge color (`dmp_Silhouette.openEdgeColor`) using R, G, B, and A values.

### Open Edge Bias Toward the Viewpoint

Use the `glUniform1fv()` function to set the bias toward the viewpoint (`dmp_Silhouette.openEdgeDepthBias`). A negative value indicates movement toward the viewpoint, and a positive value indicates movement away from it. Normal vectors are not used to generate open edges, so this bias value adjusts their appearance.

### Multiplying an Open Edge's w Component

You can multiply a vertex's w component by using the open edge's width and bias values. Use the `glUniform1i()` function to set `dmp_Silhouette.openEdgeWidthScaleByW` and `dmp_Silhouette.openEdgeDepthBiasScaleByW` to `GL_TRUE` or `GL_FALSE` for each setting.

**Table 9-3. Reserved Uniforms for Silhouette Shaders**

| Reserved Uniform | Type | Value to Set |
|---|---|---|
| `dmp_Silhouette.width` | `vec2` | Specifies the silhouette edge width using the following expression.<br>(xscale_f,<br>xscale_f * viewport.width / viewport.height)<br>xscale_f is the factory to multiply by the normal vector's x component. |
| `dmp_Silhouette.scaleByW` | `bool` | Specifies whether to apply a vertex's w component to silhouette edges.<br>Specify `GL_TRUE` to apply the vertex's w component, and `GL_FALSE` otherwise. |
| `dmp_Silhouette.color` | `vec4` | Specifies the silhouette edge color. |
| `dmp_Silhouette.frontFaceCCW` | `bool` | Specifies how a polygon is determined to be front-facing or back-facing.<br>Specify `GL_TRUE` to set CCW (counterclockwise is front-facing), and `GL_FALSE` to set CW (clockwise is front-facing). |
| `dmp_Silhouette.acceptEmptyTriangles` | `bool` | Specifies whether to render open edges.<br>Specify `GL_TRUE` to render open edges, and `GL_FALSE` otherwise. |
| `dmp_Silhouette.openEdgeColor` | `vec4` | Specifies the open edge color. |
| `dmp_Silhouette.openEdgeWidth` | `vec4` | Specifies the open edge width using the following expression.<br>(viewport.width / silhouette.width,<br>viewport.height / silhouette.width,<br>viewport.width / viewport.height,<br>2 / silhouette.width) |
| | | Specifies the open edge bias toward the viewpoint. |

| | | |
|---|---|---|
| dmp_Silhouette.openEdgeDepthBias | float | A negative value indicates movement toward the viewpoint, and a positive value indicates movement away from it. |
| dmp_Silhouette.<br>openEdgeWidthScaleByW | bool | Specifies whether to multiply the open edge width by a vertex's w component.<br>Specify GL_TRUE to multiply the open edge width by a vertex's w component, and GL_FALSE otherwise. |
| dmp_Silhouette.<br>openEdgeDepthBiasScaleByW | bool | Specifies whether to multiply a vertex's w component by the open edge bias toward the viewpoint.<br>Specify GL_TRUE to multiply the open edge width by a vertex's w component, and GL_FALSE otherwise. |

## 9.3.4. Vertex Shader Settings

A silhouette shader requires three items to render silhouette edges: vertex coordinates, vertex colors, and normal vectors. The vertex shader outputs the vertex coordinates, vertex color, and then the normal vector in that order, starting with the smallest output register number.

The output vertex attributes must be set using the vertex coordinates in the `position` attribute, the vertex color in the `color` attribute, and the normal vector in the `generic` attribute.

To output silhouette triangles, link the `DMP_silhouetteTriangle.obj` shader file, and use the following `#pragma output_map` statements in the vertex shader.

**Code 9-4. Sample Output Register Settings When a Silhouette Shader Is Used (in Shader Assembly)**

```
#pragma output_map ( position , o0 )
#pragma output_map ( color , o1 )
#pragma output_map ( generic , o2 )
```

The vertex shader runs a modelview transformation on the normal vectors input from the application. Normalized values must be output for the x and y components. In other words, the vertex shader must output a normal vector that is normalized as follows from the normal vector (nx, ny, nz) in the viewpoint coordinate system.

$$n' = n \Big/ \sqrt{nx^2 + ny^2}$$

This is shown by the following shader assembly code. *aNormal* specifies the normal vector input from the application, and *vNormal* specifies the normal vector output to the silhouette shader.

**Code 9-5. Normalizing Input Normal Vectors for the Silhouette Shader (in Shader Assembly)**

```
mov     TEMP_NORM,      CONST_0
dp3     TEMP_NORM.x,    aNormal,        MATRIX_ModelView[0]
dp3     TEMP_NORM.y,    aNormal,        MATRIX_ModelView[1]
mul     TEMP,           TEMP_NORM,      TEMP_NORM
add     TEMP,           TEMP.x,         TEMP.y
rsq     TEMP,           TEMP.x
mul     vNormal,        TEMP_NORM,      TEMP
```

To output silhouette strips, use the same procedure as you would for silhouette triangles, but link the `DMP_silhouetteStrip.obj` shader file instead.

## 9.3.5. Input Vertex Data

To render silhouette edges with the silhouette shader, call the `glDrawElements` or `glDrawArrays()` function, and specify `GL_GEOMETRY_PRIMITIVE_DMP` for the *mode* parameter. You must also disable culling by calling `glDisable(GL_CULL_FACE)` before rendering.

We also recommend using the vertex indices to input vertex data to the silhouette shader, in consideration of TWN characteristics. The following description assumes that you are using the vertex indices. If you are not using the vertex indices, you must order the vertex data according to the TWN vertex input rules.

## 9.3.6. Silhouette Triangle Indices

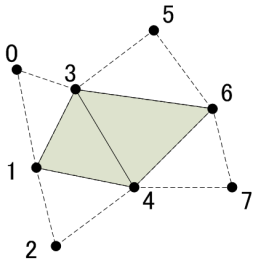
Silhouette triangles are input to the shader, one TWN at a time, using six vertices per TWN.

Vertices are input in the following order.

1. The first vertex of the center triangle.
2. The second vertex of the center triangle.
3. The remaining vertex of the adjacent triangle that shares the edge created by the first and second vertices of the center triangle.
4. The third vertex of the center triangle.
5. The remaining vertex of the adjacent triangle that shares the edge created by the first and third vertices of the center triangle.
6. The remaining vertex of the adjacent triangle that shares the edge created by the second and third vertices of the center triangle.

The object in the following figure shows how to specify indices.

**Figure 9-5. Specifying Indices With TWNs**




A setting of CCW is assumed for determining polygon facing.

Having selected the center triangles (1,4,3) and (3,4,6) so that they are front-facing, specify the indices as follows.

Triangle (1,4,3): 1, 4, 2, 3, 0, 6.
Triangle (3,4,6): 3, 4, 1, 6, 5, 7.

.

Silhouette edges are not generated for a degenerate center triangle. Triangles that have edges shared by three or more triangles were not considered.

## 9.3.7. Silhouette Strip Indices

Silhouette strips are input to the shader as consecutive TWNs, using six vertices for the first TWN and two vertices for each subsequent TWN. You can expect this to be at least twice as efficient as using silhouette triangles to render the same model.

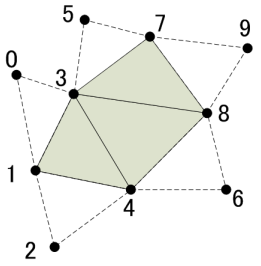
Silhouette strips can be used as input when the second and third vertices of a center triangle share an edge with another vertex to form the center triangle of the next TWN. In other words, the last adjacent triangle of a TWN is the center triangle of the next TWN.

Vertices are input in the following order.

1. The first six vertices are the same as those used for silhouette triangles. The second and third vertices of the center triangle and the last specified vertex represent the first, second, and third vertices of the next center triangle.
2. The remaining vertex of the adjacent triangle shares the edge created by the first and third vertices of the center triangle.
3. The remaining vertex of the adjacent triangle that shares the edge created by the second and third vertices of the center triangle.
4. The second and third vertices of the center triangle and the vertex specified in step 3 form vertices 1 through 3 of a new center triangle. Repeat steps 2 and 3.

To stop entering silhouette strips or to enter the next silhouette strip, give the third vertex of the center triangle as input before, the vertex specified in step 3.

**Figure 9-6. Sample Indices for Silhouette Strips**




Assuming (3,8,7) as the final center triangle in the figure,

> the silhouette strip indices would be specified in the following order: 1, 4, 2, 3, 0, 8, 6, 7, 5, **7**, 9.

The number in bold, **7**, is the vertex used to indicate the end of the silhouette strip.

To continue entering silhouette strips after you have specified the end of one strip, you must first enter the six vertices that start a new TWN. If the first center triangle of the new silhouette strip faces in the opposite direction than the one specified using `glFrontFace`, enter its first vertex twice. In other words,

> if the silhouette strip in the previous example was initially back-facing, its indices would be specified in the following order: 1, 1, 4, 2, 3, 0, 8, 6, 7, 5, 7, 9.

.

The end of a silhouette strip is specified as a delimiter when inputting multiple silhouette strips. However, if the end of the last silhouette strip is not specified, further input triangles connected to the silhouette strip may cause duplicate silhouette edges to be rendered. The end of each input silhouette strip must be specified, when alpha blending is used on silhouettes.

Note that a degenerate center triangle is considered to specify the end of a strip.

## 9.3.8. Open Edges

An open edge has already been explained to be an edge of the center triangle that is not shared with any other triangle. Because it does not share an edge with any other triangle, its indices are specified by using the remaining vertex from the center triangle. You can visualize an open edge as an adjacent triangle that has been folded precisely over the center triangle.

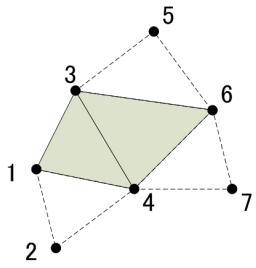
The indices for the object in Figure 9-5 would be specified as follows, if vertex 0 did not exist.

The indices (1, 4, 3) for the silhouette triangle would be specified in the order: 1, 4, 2, 3, **4**, 6.

The silhouette strip indices would be specified in the following order: 1, 4, 2, 3, **4**, 6, 5, 6, 7.

Open edges must be configured differently than silhouette edges. For more information, see 9.3.3. Reserved Uniforms.

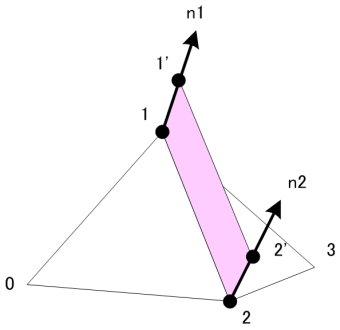
**Figure 9-7. Sample Indices for Open Edges**




## 9.3.9. Generating Silhouette Edges

A silhouette edge is rendered by generating a new rectangular polygon on the edge of a front-facing center triangle and a back-facing adjacent triangle in a TWN. Two vertices are added along the normal vectors (*n*1 and *n*2) of the two vertices (1 and 2) shared by the center and adjacent triangle to generate a rectangular polygon (from two triangular polygons).

**Figure 9-8. Rectangular Polygon That Forms a Silhouette Edge**




The coordinates (x', y', z', w') of the vertices to add are calculated by the following equation, where (x, y, z, and w) are the coordinates of a vertex on the center triangle, and (nx, ny, nz) represent a normal vector.

x' = x + x_factor * nx * w_scale
y' = y + y_factor * ny * w_scale
z' = z
w' = w

The reserved uniform for silhouette edge width sets the values applied to `x_factor`, `y_factor`, and `w_scale`.

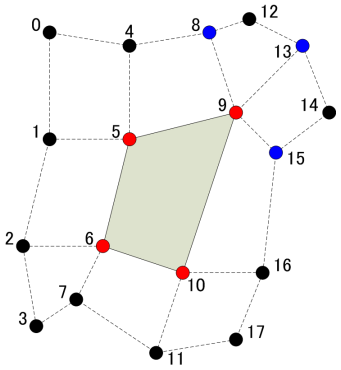
# 9.4. Catmull-Clark Subdivision Shaders

The Catmull-Clark subdivision shader uses quadrilateral polygons and their surrounding vertices to split groups of vertices into smooth polygons. **Note:** The word *subdivision* always indicates Catmull-Clark subdivision within this section.

## 9.4.1. Subdivision Patches

To subdivide polygons, give the shader **a set of polygons comprising only quadrilaterals** (a Catmull-Clark subdivision patch, or *subdivision patch* for short). A subdivision patch is made up of the target (center) quadrilateral and the group of quadrilaterals that share edges formed by that center quadrilateral's four vertices.

**Figure 9-9. Sample Catmull-Clark Subdivision Patch**




A subdivision patch **can only be applied to a polygon model that is entirely made up of quadrilaterals**.

Each vertex in the central quadrilateral usually forms four edges, like vertices 5, 6, and 10. However, a vertex that forms three or five edges like vertex 9 is called an *extraordinary point*, and its edge count is called its *valence*. **A subdivision patch can only have one extraordinary point. It must be in the central quad and have a valence from 3 through 12.**

If a subdivision patch's central quad has an extraordinary point, you must start specifying indices from that point. Any other subdivision patches with central quads that have the same extraordinary point must be entered consecutively. Holes will appear in the mesh in some cases.

## 9.4.2. Shader Files

The shader file to link with the vertex shader is determined by the number of vertex attributes output for fragment processing that are not required by the subdivision shader.

Subdivision: DMP_subdivision*N*.obj

Where *N* is the number of vertex attributes (1 through 6) that are not required by the subdivision shader.

## 9.4.3. Reserved Uniform

Subdivision shaders have the following reserved uniforms. These reserved uniforms must be set by the application because they initially have undefined values.

#### Subdivision Level

Use the `glUniform1f()` function to set the subdivision level (`dmp_Subdivision.level`), which controls how finely the shader subdivides polygons. Higher levels (larger numbers) indicate finer subdivision. At the smallest value of 0, a single vertex is added to the center of the subdivision patch, and the patch's original vertex coordinates are adjusted.

#### Using Quaternions

The subdivision shader generates new output vertices that have interpolated vertex attributes for their attributes that are not vertex coordinates. Because quaternions must be subdivided in a particular way, the shader must be notified when it is given quaternions.

Use the `glUniform1i()` function to either enable (`GL_TRUE`) or disable (`GL_FALSE`) quaternions (`dmp_Subdivision.fragmentLightingEnabled`).

**Table 9-4. Reserved Uniforms for the Catmull-Clark Subdivision Shader**

| Reserved Uniform | Type | Value to Set |
|---|---|---|
| `dmp_Subdivision.level` | `float` | Specifies the subdivision level.<br>0 (The lowest subdivision level).<br>1<br>2 (The highest subdivision level). |
| `dmp_Subdivision.fragmentLightingEnabled` | `bool` | Specifies whether to use quaternions required for fragment lighting.<br>Specify `GL_TRUE` to use quaternions., and `GL_FALSE` otherwise. |

## 9.4.4. Vertex Shader Settings

The subdivision shader requires one vertex attribute: the vertex coordinates. The vertex shader outputs the vertex coordinates, starting with the smallest output register number.

Two output vertex attributes must be set: the `position` attribute of the vertex coordinates and one other attribute.

If the vertex shader outputs the vertex color in addition to the vertex coordinates required by the subdivision shader, link the `DMP_subdivision1.obj` shader file and use the following `#pragma output_map` statements in the vertex shader.

**Code 9-6. Sample Output Register Settings When the Catmull-Clark Subdivision Shader Is Used (in Shader Assembly)**

```
#pragma output_map ( position , o0 )
#pragma output_map ( color , o1 )
```

When quaternions are required for fragment lighting, they must be output to the register with the smallest number following the register that outputs the vertex coordinates.

**Code 9-7. Sample Output Register Settings When the Subdivision Shader Uses Quaternions (in Shader Assembly)**

```
#pragma output_map ( position , o0 )
#pragma output_map ( quaternion , o1 )
#pragma output_map ( color , o2 )
```

## 9.4.5. Input Vertex Data

To use Catmull-Clark subdivision, call the `glDrawElements()` function, and specify `GL_GEOMETRY_PRIMITIVE_DMP` for the *mode* parameter. You cannot use the `glDrawArrays()` function. Vertex indices must also be used through the vertex buffer.

## 9.4.6. Subdivision Patch Indices

The number of vertices in a subdivision patch depends on the valence (from 3 through 12) of its extraordinary point. A subdivision patch does not have a fixed number of input vertices.

Enter the size of the subdivision patch first. Specify the size of the subdivision patch as the number of vertices that it includes. To find this number, double the extraordinary point's valence, and add 8. A subdivision patch's size must be in the range from 14 through 32. It is 16 when the patch does not have an extraordinary point. Behavior is undefined for input patch sizes larger than 32.

Specify a subdivision patch's indices in the following order.

1. The four vertices in the central quad (following the order specified when `glFrontFace` was called, starting from the extraordinary point, if it exists).
2. The vertices around the central quad (following the order specified when `glFrontFace` was called).

Assuming that polygons with a counterclockwise (CCW) winding are front-facing, the indices for the subdivision patch in Figure 9-8 would be specified in the following order.

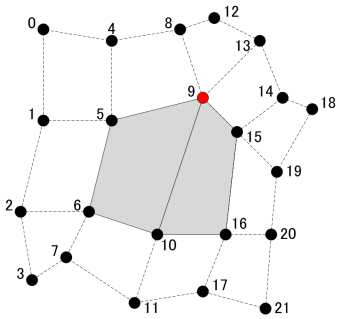
Subdivision Patch Indices: **18**, 9, 5, 6, 10, 8, 4, 0, 1, 2, 3, 7, 11, 17, 16, 15, 14, 13, 12.

. The first number, 18, is the size of the subdivision patch.

It is followed by the subdivision patch for the central quad, with the extraordinary point's vertex 9. In the following figure, when subdivision patch indices are used,

the resulting subdivision patch indices would be **18**, 9, 10, 16, 15, 5, 6, 7, 11, 17, 21, 20, 19, 18, 14, 13, 12, 8, 4.

**Figure 9-10. Sample Subdivision Patch Indices**



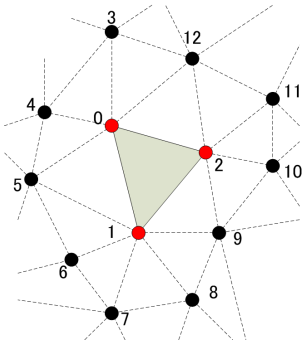
# 9.5. Loop Subdivision Shader

The Loop subdivision shader uses triangular polygons and their surrounding vertices to split groups of vertices into smooth polygons. The word *subdivision* always indicates Loop subdivision within this section.

## 9.5.1. Subdivision Patches

To subdivide polygons, give the shader a Loop subdivision patch (referred to later simply as a *subdivision patch*). A subdivision patch comprises the target (center) triangle and the group of vertices that share edges with that triangle's three vertices.

**Figure 9-11. Sample Loop Subdivision Patch**




Each vertex in the center triangle forms a number of edges that is called its *valence*.

With the vertices 0, 1, 2 in Figure 9-11 set as the center triangle, the valence of vertex 0 is 6, the valence of vertex 1 is 7, and the valence of vertex 2 is 6. **The valence of each patch that accepts a subdivision patch is 3 to 12, and the total of the valences of the three vertices of the center triangle must be 29 or less** .

You can increase the number of virtual vertices to include a vertex with a valence of 2 (a vertex that only shares an edge with other vertices in the center triangle) in a subdivision patch.

## 9.5.2. Shader Files

The number of output registers configured to send non-valence vertex attributes to the subdivision shader determines which shader file to link with the vertex shader. There must be at least one output register because vertex coordinates are required output.

Subdivision: DMP_subdivision*N*.obj

Where *N* is the number of output registers (1 through 4) configured with vertex attributes, other than the valence.

## 9.5.3. Reserved Uniform

The Loop subdivision shader has the same reserved uniforms as the Catmull-Clark subdivision shader. For more information, see 9.4.3. Reserved Uniforms. The reserved uniform for the loop

subdivision shader must be set by the application because its initial value is undefined.

Although new vertices are not added when the subdivision level is 0, the subdivision patch's original vertex coordinates are adjusted.

## 9.5.4. Vertex Shader Settings

The subdivision shader requires two attributes: the vertex coordinates and the valence. The vertex shader outputs these attributes, starting at the smallest output register number. The vertex coordinates are first, followed by any other vertex attributes that exist, and finally, followed by the valence.

Two output vertex attributes must be set: the vertex coordinates using the `position` attribute, and the valence using the `generic` attribute.

If the vertex shader outputs the vertex color in addition to the vertex coordinates required by the subdivision shader, link the `DMP_loopSubdivision2.obj` shader file (because the vertex coordinates are also included in the number of output registers) and use the following `#pragma output_map` statements in the vertex shader.

**Code 9-8. Sample Output Register Settings When the Loop Subdivision Shader Is Used (in Shader Assembly)**

```
#pragma output_map ( position , o0 )
#pragma output_map ( color , o1 )
#pragma output_map ( generic, o2 )
```

When quaternions are required for fragment lighting, they must be output to the register with the smallest number following the register that outputs the vertex coordinates. Because the number of non-valence output registers is restricted to 4 or less, multiple attributes must be packed into a single register, when there are five or more vertex attributes other than the valence. Quaternions, however, cannot be packed with other vertex attributes.

**Code 9-9. Sample Output Register Settings When the Subdivision Shader Uses a Large Number of Vertex Attributes (in Shader Assembly)**

```
#pragma output_map ( position , o0 )
#pragma output_map ( quaternion , o1 )
#pragma output_map ( color , o2 )
#pragma output_map ( texture0 , o3.xy )
#pragma output_map ( texture1 , o3.zw )
#pragma output_map ( generic , o4 )
```

## 9.5.5. Input Vertex Data

To use Loop subdivision, call the `glDrawElements()` function and, for the *mode* parameter, pass `GL_GEOMETRY_PRIMITIVE_DMP`. You cannot use the `glDrawArrays()` function. Vertex indices must also be used through the vertex buffer.

## 9.5.6. Subdivision Patch Indices

The number of vertices in a subdivision patch depends on the total valence of the center triangle. A subdivision patch does not have a fixed number of input vertices.

Enter the size of the subdivision patch first. Specify the size of the subdivision patch as three plus the total valence of the vertices that form the center triangle.

Specify a subdivision patch's indices in the following order.

1. The indices of the three vertices (v0, v1, and v2) that form the center triangle (following the order specified when `glFrontFace` was called).
2. All vertices that share an edge with v0 (in any order, although, the same order must also be used by any other subdivision patch that includes the same vertex).
3. All vertices that share an edge with v1 (in any order, although, the same order must also be used by any other subdivision patch that includes the same vertex).
4. All vertices that share an edge with v2 (in any order, although, the same order must also be used by any other subdivision patch that includes the same vertex).
5. A fixed value of 12 and the center triangle's three vertices (12, v0, v1, and v2).
6. A vertex (e00) that forms a triangle with v0 and v2 and is not in the center triangle.
7. A vertex (e10) that forms a triangle with v0 and v1 and is not in the center triangle.
8. A vertex (e20) that forms a triangle with v1 and v2 and is not in the center triangle.
9. The vertex that shares an edge with v0 and is **next to e00, in counterclockwise order**.
10. The vertex that shares an edge with v1 and is **next to e10, in counterclockwise order**.
11. The vertex that shares an edge with v2 and is **next to e20, in counterclockwise order**.
12. The vertex that shares an edge with v0 and is **next to e10, in clockwise order**.
13. The vertex that shares an edge with v1 and is **next to e20, in clockwise order**.
14. The vertex that shares an edge with v2 and is **next to e00, in clockwise order**.

Assuming that polygons with a counterclockwise (CCW) winding are front-facing, the indices for the subdivision patch in Figure 9-11 would be specified in the following order.

Subdivision Patch Indices:

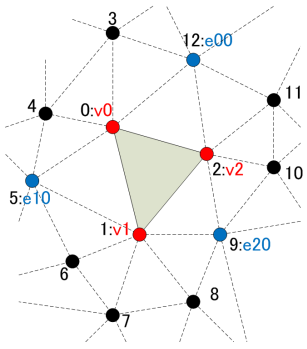
**22**, 0, 1, 2, 1, 2, 12, 3, 4, 5, 2, 0, 5, 6, 7, 8, 9, 0, 1, 9, 10, 11, 12,
**12**, 0, 1, 2, 12, 5, 9, 3, 6, 10, 4, 11, 8.

. The number 22 on the first line is the size of the subdivision patch. The number 12 on the second line is a fixed value.

When another subdivision patch uses v0 in its center triangle, the vertices that share an edge with it must be specified in the same order: (1, 2, 12, 3, 4, 5). The same applies to v1 and v2.

Although some vertices will be specified more than once when specifying a subdivision patch, they are read from the cache after vertex processing, and do not actually impose a performance penalty.

**Figure 9-12. Sample Subdivision Patch Indices**



# 9.6. Particle System Shaders

Particle system shaders are used by particle systems that render a large number of point sprites (particles) along a Bézier curve.

Particles are rendered along a Bézier curve that is defined by four control points input to the shader. Each control point is randomly placed within its own bounding box, changing the Bézier curve.

A particle's color, size, angle of texture coordinate rotation, and other attributes are interpolated based upon its position on the Bézier curve.

## 9.6.1. Shader Files

Link the vertex shader to the correct particle system shader file, based on the features that you want to support.

Particle system shader: DMP_particleSystem_*X*_*X*_*X*_*X*.obj

Where each *X* represents a value of `0` or `1` that controls a particle system feature. These features are, in order, particle time clamping, texture coordinate rotation, the use of RGBA components or the alpha component alone, and output of texture coordinate 2. A feature is not necessarily disabled by a value of 0, nor enabled by a value of 1. Refer to the following table to determine which shader file to link.

**Table 9-5. Shader Filenames and the Particle System Features They Support**

| Filename | Time Clamping | Texture Coordinate Rotation | RGBA Colors | Texture coordinate 2 |
|---|---|---|---|---|
| *_0_0_0_0.obj | Yes | Yes | (Alpha only) | No |
| *_0_0_0_1.obj | Yes | Yes | (Alpha only) | Yes |
| *_0_0_1_0.obj | Yes | Yes | Yes | No |
| *_0_0_1_1.obj | Yes | Yes | Yes | Yes |
| *_0_1_0_0.obj | Yes | None | (Alpha only) | No |
| *_0_1_0_1.obj | Yes | None | (Alpha only) | Yes |
| *_0_1_1_0.obj | Yes | None | Yes | No |
| *_0_1_1_1.obj | Yes | None | Yes | Yes |
| *_1_0_0_0.obj | None | Yes | (Alpha only) | No |
| *_1_0_0_1.obj | None | Yes | (Alpha only) | Yes |
| *_1_0_1_0.obj | None | Yes | Yes | No |
| *_1_0_1_1.obj | None | Yes | Yes | Yes |
| *_1_1_0_0.obj | None | None | (Alpha only) | No |
| *_1_1_0_1.obj | None | None | (Alpha only) | Yes |
| *_1_1_1_0.obj | None | None | Yes | No |
| *_1_1_1_1.obj | None | None | Yes | Yes |

Each asterisk (*) in the table stands for `DMP_particleSystem`.

## 9.6.2. Reserved Uniform

Particle system shaders have the following reserved uniforms. These reserved uniforms must be set by the application because they initially have undefined values.

### Color

Use the `glUniformMatrix4fv()` function to set the particle color (`dmp_PartSys.color`). A 4x4 matrix represents the particle color of the first, second, third, and fourth control point, using an RGBA value in each of the corresponding rows. This setting is valid only for a shader file that uses RGBA colors (`DMP_particleSystem_X_X_1_X.obj`).

$$color = \begin{pmatrix} R_1 & G_1 & B_1 & A_1 \\ R_2 & G_2 & B_2 & A_2 \\ R_3 & G_3 & B_3 & A_3 \\ R_4 & G_4 & B_4 & A_4 \end{pmatrix}$$

Performance is worse when RGBA colors are used than when alpha components alone are used. If you are not using color components, we recommend that you link to a shader file that uses alpha components alone (`DMP_particleSystem_X_X_0_X.obj`).

### Aspect

Use the `glUniformMatrix4fv()` function to set the particle aspect (`dmp_PartSys.aspect`) with a 4x4 matrix. Each row corresponds to a control point (from 1 through 4) and configures the particle size, texture coordinate rotation, texture coordinate scaling, and alpha component.

$$aspect = \begin{pmatrix} size_1 & rotate_1 & scale_1 & alpha_1 \\ size_2 & rotate_2 & scale_2 & alpha_2 \\ size_3 & rotate_3 & scale_3 & alpha_3 \\ size_4 & rotate_4 & scale_4 & alpha_4 \end{pmatrix}$$

### Size

Set the particle size (the first aspect column) to a value of 1.0 or greater.

Use the `glUniform2fv()` function to set the minimum and maximum particle size (`dmp_PartSys.pointSize`). Use the `glUniform2fv()` function to set the reciprocal of the viewport's width and height (`dmp_PartSys.viewport`) because particle rendering does not account for the screen size. If distance attenuation is being applied to the particle size, call the `glUniform3fv()` function to set the distance attenuation factor (`dmp_PartSys.distanceAttenuation`), and specify the attenuation coefficients that calculate the attenuated size, as shown in the following equation.

$$derived\_size = size \times \sqrt{\frac{1}{a + b \times d + c \times d^2}}$$

*derived_size* is the size with distance attenuation applied, *size* is the original size, and *d* is the distance from the viewpoint.

### Texture Coordinates

You can set the rotation (the second aspect column) and scaling (the third aspect column) of texture coordinates at each control point.

The particle system outputs two texture coordinates: 0 and 2. Both texture coordinates support rotations, but only texture coordinate 2 supports scaling. The linked shader file enables or disables these settings.

- Output texture coordinate 2 (`DMP_particleSystem_X_X_X_1.obj`, do not output (`DMP_particleSystem_X_X_X_0.obj`)
- Rotate and scale the texture coordinate (`DMP_particleSystem_X_0_X_X.obj`), neither rotate nor scale (`DMP_particleSystem_X_1_X_X.obj`)

Rotations are specified in radians. Clockwise rotations are specified by positive values.

Texture coordinate 0 is output with the particle's lower-left, lower-right, upper-left, and upper-right corners at (0,0), (1,0), (0,1), and (1,1) respectively. Texture coordinate 2 is output with the particle's lower-left, lower-right, upper-left, and upper-right corners at (-1,-1), (1,-1), (-1,1), and (1,1) respectively.

Given a rotation of A and a scaling value of R, texture coordinate 0 is calculated as follows.

Lower-left = ( 0.5×(1.0 + (-cosA + sinA)), 0.5×(1.0 + (-cosA - sinA)) )
Lower-right = ( 0.5×(1.0 + (cosA + sinA)), 0.5×(1.0 + (-cosA + sinA)) )
Upper-left = ( 0.5×(1.0 + (-cosA - sinA)), 0.5×(1.0 + (cosA - sinA)) )
Upper-right = ( 0.5×(1.0 + (cosA - sinA)), 0.5×(1.0 + (cosA + sinA)) ).

Texture coordinate 2 is calculated as follows.

Lower-left = ( R×(-cosA + sinA), R×(-cosA - sinA) )
Lower-right = ( R×(cosA + sinA), R×(-cosA + sinA) )
Upper-left = ( R×(-cosA - sinA), R×(cosA - sinA) )
Upper-right = ( R×(cosA - sinA), R×(cosA + sinA) ).

.

### Alpha Component

Set the particle's alpha component (the fourth aspect column) as a value between 0.0 and 1.0. This setting is used with the linked shader files that only use the alpha component (`DMP_particleSystem_X_X_0_X.obj`).

### Emission Count

Use the `glUniform1fv()` function to set the maximum particle emission count (`dmp_PartSys.countMax`). Set this value to one less than the actual number of particles you want to emit. You must set a value of `0.0` or greater. However, because the shader program implementation limits the maximum number of emitted particles to 255, no more than 255 particles will be emitted, even when this reserved uniform is set to a value greater than 256.

### Execution Time and Speed

A particle system has a concept of time. Use the `glUniform1fv()` function to set the particle system time (`dmp_PartSys.time`) to the current time. This current time is randomly converted into each particle's execution time, during which the particle travels from the first control point to the fourth control point. A particle is emitted at the first control point when its execution time is 0.0, and reaches the fourth control point when its execution time is 1.0.

If you link to a shader file that clamps the execution time (`DMP_particleSystem_0_X_X_X.obj`), particles with an execution time of 1.0 or greater cease to be rendered. Consequently, particles will cease to be emitted at some point, if the application simply lets time pass without resetting the execution time.

If you link to a shader file that does not clamp the execution time (`DMP_particleSystem_1_X_X_X.obj`), the execution time loops between 0.0 and 1.0. In other words, particles that reach the fourth control point are re-emitted from the first control point.

Use the `glUniform1fv()` function to set the particle speed (`dmp_PartSys.speed`).

### Random Values

The position of control points within their bounding boxes and the execution time of particles are determined using a function that generates pseudorandom numbers. The application can specify a random seed and coefficient to use in this random function.

The implementation of the random function is similar to the following algorithm for a pseudorandom number generator.

$$X_{N+1} = (aX_N + b) \bmod m$$

Use the `glUniform4fv()` function to set the random seed (`dmp_PartSys.randSeed`) with an array of values (the x, y, and z components of the Bézier curve and the particle execution time, in that order) corresponding to $X_0$ in Equation 9-8.

For the random function's coefficients (`dmp_PartSys.randomCore`), use the `glUniform4fv()` function to specify the values a, b, m, and 1/m in Equation 9-8.

**Table 9-6. Reserved Uniforms for Particle System Shaders**

| Reserved Uniform | Type | Value to Set |
|---|---|---|
| `dmp_PartSys.color` | `mat4` | The color at each control point.<br>(R, G, B, A) * 4vec<br>Each component has a value in the range from 0.0 through 1.0. |
| `dmp_PartSys.aspect` | `mat4` | The aspect of each control point.<br>(particle_size, rotation_angle, scale, alpha) * 4vec<br>*particle_size* is `1.0` or greater and *alpha* is in the range from `0.0` through `1.0`. |
| `dmp_PartSys.time` | `float` | The current particle system time. |
| `dmp_PartSys.speed` | `float` | The speed of particle movement.<br>0.0 or more |
| `dmp_PartSys.countMax` | `float` | A number that is one less than the maximum number of particles to emit.<br>0.0 or more |
| `dmp_PartSys.randSeed` | `vec4` | The random seed for each of the random functions.<br>(Multiplicand with the Bézier curve's x component, multiplicand with the Bézier curve's y component, multiplicand with the Bézier curve's z component, multiplicand with the particle execution time). |
| `dmp_PartSys.randomCore` | `vec4` | The random function's coefficients.<br>(a, b, m, 1/m) |
| `dmp_PartSys.distanceAttenuation` | `vec3` | The distance attenuation factor. |
| `dmp_PartSys.viewport` | `vec2` | The viewport using the following equation.<br>(1 / viewport.width, 1 / viewport.height) |
| `dmp_PartSys.pointSize` | `vec2` | The minimum and maximum particle size.<br>Each has a value of `0.0` or greater. |

## 9.6.3. Vertex Shader Settings

A particle system shader requires a 4×4 matrix that contains the following values converted into clip coordinates: a single control point's vertex coordinates and the radius for the x, y, and z components of its bounding box (centered on said vertex coordinates). Starting at the smallest output register number, the vertex shader outputs the vertex coordinates followed by the first, second, third, and fourth rows of the conversion matrix, in that order.

Two output vertex attributes must be set: the vertex coordinates using the `position` attribute and the converted matrix using the `generic` attribute.

**Code 9-10. Sample Output Register Settings When a Particle System Shader Is Used (in Shader Assembly)**

```
#pragma output_map ( position , o0 )
#pragma output_map ( generic , o1 )
#pragma output_map ( generic , o2 )
#pragma output_map ( generic , o3 )
#pragma output_map ( generic , o4 )
```

The following equation shows the conversion into clip coordinates. The radii for the bounding box's x, y, and z components are: *Rx*, *Ry*, and *Rz* respectively. The projection matrix is *Mproj* and the modelview matrix is *Mmodelview*.

$$\begin{pmatrix} o1.x & o1.y & o1.z & 0 \\ o2.x & o2.y & o2.z & 0 \\ o3.x & o3.y & o3.z & 0 \\ o4.x & o4.y & o4.z & 0 \end{pmatrix} = Mproj \times Mmodelview \times \begin{pmatrix} Rx & 0 & 0 & 0 \\ 0 & Ry & 0 & 0 \\ 0 & 0 & Rz & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

This is shown by the following shader assembly code. `aBoundingBox` is a vector with the x, y, and z components of the radius input by the application, and `vBoundingBox1` through `vBoundingBox4` represent the matrix output to the particle system shader. The bounding box's radius is input as attributes in this sample code, but because the particle system shader requires data for only four vertices, you can use an implementation that sets all of this along with the vertex coordinates in uniforms.

**Code 9-11. Bounding Box Radius and Clip Coordinate Conversion (in Shader Assembly)**

```
mov     TEMP_BOX[0],    CONST_0
mov     TEMP_BOX[1],    CONST_0
mov     TEMP_BOX[2],    CONST_0
mov     TEMP_BOX[3],    CONST_0
mov     TEMP_BOX[0].x,  aBoundingBox.x
mov     TEMP_BOX[1].y,  aBoundingBox.y
mov     TEMP_BOX[2].z,  aBoundingBox.z
m4x4    TEMP_MAT,       MATRIX_Project, MATRIX_ModelView
m4x4    TEMP_MAT,       TEMP_MAT,       TEMP_BOX
mov     vBoundingBox1,  TEMP_MAT[0]
mov     vBoundingBox2,  TEMP_MAT[1]
mov     vBoundingBox3,  TEMP_MAT[2]
mov     vBoundingBox4,  TEMP_MAT[3]
```

## 9.6.4. Input Vertex Data

Calls from the `glDrawArrays()` function are not supported. When using a particle system shader, call the `glDrawElements()` function, and specify `GL_GEOMETRY_PRIMITIVE_DMP` for the *`mode`* parameter.

# 10. Rasterization

Even primitives created by geometry shaders, such as the point and line shader, are all ultimately converted to triangle primitives and then generated (involving triangle generation and triangle setup). The generated triangles are culled, clipped, converted into window coordinates, and finally rasterized into a collection of fragments. Unlike in OpenGL ES 2.0, the scissor test is performed during the rasterization stage. Further processing affects the generated fragments.

**Figure 10-1. Process From the Vertex and Geometry Shaders to Rasterization**

All processing after rasterization is implemented as a fixed pipeline, so there are fixed vertex attributes that can be assigned to fragments. The following are the major vertex attributes.

- Window coordinates
- Depth values
- Texture coordinates and partial differential values
- Quaternions
- View vectors
- Vertex colors (rasterized from absolute values).

## 10.1. Culling

The order in which vertices are specified for the generated triangles (polygons) determines which face, front or back, is facing the viewer. Culling is a feature that determines whether to rasterize a polygon based on whether it is back-facing.

### 10.1.1. Determining the Front and Back Faces

A polygon's front and back faces are determined by the order in which its triangles' vertices are specified in window coordinates. Either a clockwise (CW) or a counterclockwise (CCW) winding can be considered to be front-facing. You can specify this using the `glFrontFace()` function.

**Code 10-1. Definition of the glFrontFace Function**

```
void glFrontFace(GLenum mode);
```

For *`mode`*, specify `GL_CW` for clockwise winding or `GL_CCW` for counterclockwise winding. A counterclockwise winding (`GL_CCW`) is configured by default.

### 10.1.2. How to Use

Culling is used in the same way it is used in OpenGL

#### Enabling or Disabling Culling

To enable or disable culling, call `glEnable` or `glDisable`, respectively, and specify `GL_CULL_FACE` for *`cap`*. Call the `glIsEnabled()` function, and specify `GL_CULL_FACE` for *`cap`* to get the current setting. Culling is disabled by default.

#### Specifying the Culled Face

Use the `glCullFace()` function to specify which face is not rasterized (the culled face).

**Code 10-2. Definition of the glCullFace Function**

```
void glCullFace(GLenum mode);
```

You can choose from the following *`mode`* values to specify the culled face.

**Table 10-1. Specifying the Culled Face**

| Setting Value | Culled Face |
|---|---|
| GL_FRONT | Front |
| GL_BACK (default) | Back |
| GL_FRONT_AND_BACK | Both |

## 10.2. Clipping

Clipping is a feature that clips (removes) primitives from the regions (clip volumes) at which the view volume intersects the half-spaces defined by the specified clipping planes. 3DS clipping features correspond to the clipping features in OpenGL ES 1.1, but they are all controlled using reserved uniforms. The clipping implementation also generates new vertices and triangles for each clipped triangle, splitting it into multiple triangles.

Because GPU vertex processing converts coordinates using 24-bit floating-point numbers, clipping may not be performed correctly at the far clipping plane, if the ratio of the near clipping plane to the far clipping plane is large. Either configure the clip volume to keep the ratio of the near clipping plane to the far clipping plane small, or avoid placing polygons near the far clipping plane, whenever possible.

### 10.2.1. Reserved Uniform

The following reserved uniforms are used for clipping.

#### Enabling or Disabling Clipping

To enable clipping, set the reserved uniform `dmp_FragOperation.enableClippingPlane` to `GL_TRUE`, using the `glUniform1i()` function. This setting is disabled (set to `GL_FALSE`) by default.

#### Clipping Plane

To specify the clipping plane, set four coefficients in the reserved uniform `dmp_FragOperation.clippingPlane` using the `glUniform4f()` function. If p1, p2, p3, and p4 are the four coefficients, the clip volume is a collection of points that satisfy the following equation.

$$\begin{pmatrix} p1 & p2 & p3 & p4 \end{pmatrix} \begin{pmatrix} Xc \\ Yc \\ Zc \\ Wc \end{pmatrix} \geq 0$$

Because these coefficients must be defined in the clip coordinate system, you must specify values that have been through a modelview transformation and perspective projection for the clipping plane used by the OpenGL ES standard. Unlike the OpenGL ES specifications, the 3DS system clips z-coordinates between `0` and `–Wc`. OpenGL ES-compatible matrices cannot be used unchanged for perspective projections. All coefficients are set to 0.0 by default.

> **Note:** For cautions related to using OpenGL ES-compatible matrices with projection transformations, see 8.4. Notes for the Clip Coordinate System.

**Table 10-2. Reserved Uniforms Used for Clipping**

| Reserved Uniform | Type | Value to Set |
|---|---|---|
| `dmp_FragOperation.enableClippingPlane` | `bool` | Enables or disables clipping.<br>`GL_TRUE` or `GL_FALSE` (default). |
| `dmp_FragOperation.clippingPlane` | `vec4` | Specifies the four coefficients for the clipping plane.<br>This is (0.0, 0.0, 0.0, 0.0) by default. |

## 10.3. Transformation to Window Coordinates

A polygon model's vertices are transformed from object coordinates into window coordinates through the following four steps.

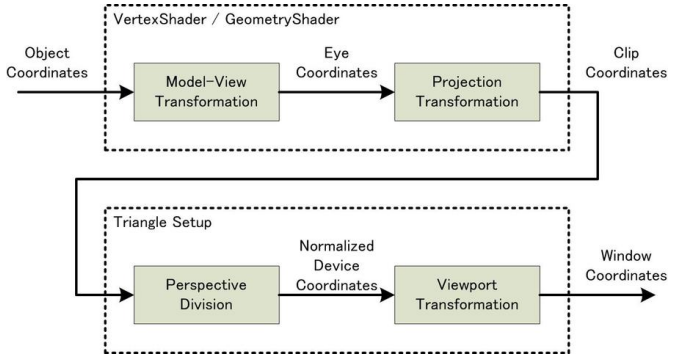
- Modelview transformation:
  Transforms object coordinates into eye coordinates.
- Projection transformation:
  Transforms eye coordinates into clip coordinates. Note that the converted values (*Zc*) are between -*Wc* and 0.
- Perspective division:
  Transforms clip coordinates into normalized device coordinates using w values.

- Viewport transformation:
  Transforms normalized device coordinates into window coordinates following the viewport settings.

The vertex shader (and geometry shader) must output vertex attributes in clip coordinates. As a result, the vertex shader generally performs modelview and projection transformations internally. Perspective division and viewport transformation determine the position of each generated triangle in the display region. These triangles are converted into fragments during rasterization, and are then used by processes such as fragment lighting.

The following figure illustrates the process of transforming coordinates.

**Figure 10-2. Process of Transforming Vertices From Object Coordinates Into Window Coordinates**




## 10.3.1. Configuring the Viewport

Normalized device coordinates are transformed into window coordinates by the following equation.

$$\begin{pmatrix} Xw \\ Yw \\ Zw \end{pmatrix} = \begin{pmatrix} (p_x / 2)Xd + o_x \\ (p_y / 2)Yd + o_y \\ n - (f - n) \times Zd \end{pmatrix}$$

*Xw*, *Yw*, and *Zw* are window coordinates.
*Xd*, *Yd*, and *Zd* are normalized device coordinates.
$p_x$ and $p_y$ are the viewport's width and height.
$o_x$ and $o_y$ are the viewport's center point.
*n* and *f* are the depth values of the near and far planes, respectively, in clip space.

You can use the `glDepthRangef()` function to set the values applied to *n* and *f* in this equation.

**Code 10-3. Definition of the glDepthRangef Function**

```
void glDepthRangef(GLclampf zNear, GLclampf zFar);
```

The values set for *zNear* and *zFar* are clamped between `0.0` and `1.0`. By default, *zNear* is `0.0` and *zFar* is `1.0`.

$p_x$, $p_y$, $o_x$, and $o_y$ can all be calculated from the viewport settings. The `glViewport()` function configures the viewport.

**Code 10-4. glViewport Function**

```
void glViewport(GLint x, GLint y, GLsizei width, GLsizei height);
```

For *x* and *y* specify the coordinates of the viewport's starting point (lower-left corner). A `GL_INVALID_VALUE` error is generated if a negative value is specified. If the specified value is not a multiple of 4, processing efficiency drops (to half for an even number and to one third for an odd number). In this case, extend the viewport so that it is a multiple of 4, adjust the perspective projection matrix so that it renders correctly on the extended viewport, and then apply the scissor test to avoid rendering unnecessary regions.

For *width* and *height*, specify the width and height of the viewport. A `GL_INVALID_VALUE` error is generated if a negative value is specified. The maximum width and height are both 1024.

The value of *width* is applied to $p_x$, *height* is applied to $p_y$, (*x* + *width*) / 2 is applied to $o_x$, and (*y* + *height*) / 2 is applied to $o_y$.

#### 10.3.1.1. A Viewport Larger Than 1023×1016 Causes Incorrect Rendering

A hardware bug prevents images from being rendered properly if the following conditions are met when the `glViewport()` function sets the viewport.

- When *width* is greater than `1023`, entire polygons are not rendered if they contain pixels whose window x-coordinate is at least 1023 greater than the *x* parameter of the `glViewport()` function (taking the left side of the window to have an x-coordinate of `0`).
- When *height* is greater than `1016`, the GPU stops responding if any polygon contains pixels whose window y-coordinate is at least 1016 greater than the *y* parameter of the `glViewport()` function (taking the bottom of the window to have a y-coordinate of `0`).

To work around this hardware bug, avoid rendering pixels at the problematic coordinates by setting the viewport's *width* and *height* to values that are not greater than `1023` and `1016`, respectively.

When using the render-to-texture technique with a 1024×1024 texture, you must only render to a 1023×1016 region, and you must adjust texture coordinates so that only a 1023×1016 texture region is valid for use.

If you want to render to the entire 1024×1024 region, you must keep the viewport's *width* from exceeding 1023 and its *height* from exceeding 1016, while changing the viewport's offset to render the texture in sections. For example, the bug described in this section does not occur if you split rendering into the following four function calls: `glViewport(0, 0, 512, 512)`, `glViewport(512, 0, 512, 512)`, `glViewport(0, 512, 512, 512)`, and `glViewport(512, 512, 512, 512)`.

You cannot work around this bug by using the scissor test to prevent the pixels at the problematic coordinates from being rendered.

### 10.3.2. Polygon Offset

Polygon offset is a feature that adds an offset to depth values when polygons are rasterized and converted into fragments. This resolves the situation in which a lack of depth value resolution prevents the fragments' front-to-back order from being determined, as for overlapping coplanar polygons. Polygon offset occurs during window coordinates conversion.

#### Enabling or Disabling Polygon Offset

To enable or disable polygon offset, call `glEnable` or `glDisable`, respectively, and specify `GL_POLYGON_OFFSET_FILL` for *`cap`*. Call the `glIsEnabled()` function, and specify `GL_POLYGON_OFFSET_FILL` for *`cap`* to get the current setting. Polygon offset is disabled by default.

#### Specifying Offset Values

You can use the `glPolygonOffset()` function to specify the offset for depth values when polygon offset is enabled.

**Code 10-5. Definition of the glPolygonOffset Function**

```
void glPolygonOffset(GLfloat factor, GLfloat units);
```

OpenGL uses the values of *`factor`* and *`units`* to determine the offset, but the 3DS system uses only *`units`*. The value of *`factor`* is set, but it is irrelevant to the offset.

The offset value is set with the product of *`units`* and the minimum (fixed) value at which a difference in the depth value appears in window coordinates. Because the z-value for vertex coordinates is implemented as a 24-bit floating-point number after vertex processing, a *`units`* value that is not a multiple of 128 has no effect when a polygon's z-value is close to 1.0. To get a definite effect, set *`units`* to a multiple of 128.

Depth values are written to the depth buffer after an offset is added to them.

## 10.3.3. W-Buffer

The w-buffer is a feature that calculates depth values in window coordinates without using a perspective projection. You can control the w-buffer with the following reserved uniforms. Settings made by the `glDepthRangef()` function are disabled when the w-buffer is enabled.

#### Depth Value Scale

Depth values are calculated by the following equation, when the w-buffer is enabled.

$$Zw = -scale_w \times Zc$$

*Zw* is the depth value in window coordinates, *Zc* is the z-value in clip coordinates, and $scale_w$ is the scale value. This scale value is a floating-point number set by the reserved uniform `dmp_FragOperation.wScale`. As long as it is not 0.0, the w-buffer is enabled. Set the scale such that $Z_w$ remains in the range from `0.0` to `1.0`.

**Table 10-3. Reserved Uniforms Used for the W-Buffer**

| Reserved Uniform | Type | Setting Value |
|---|---|---|
| `dmp_FragOperation.wScale` | `float` | Specifies the depth value scale. 0.0 by default. |

#### When Polygon Offset Is Enabled

If both the w-buffer and polygon offset features are enabled, the value obtained by multiplying $W_c$ (the w value in clip coordinates) times *units*, as specified by the `glPolygonOffset()` function, is applied to as the polygon offset value.

# 10.4. Scissor Test

The scissor test is a feature that rejects fragments outside the specified range of window coordinates to reduce the number of fragments handled by further processing.

## 10.4.1. How to Use

Although its position in the pipeline is different, the process has the same specifications as the OpenGL scissor test.

#### Enabling or Disabling the Scissor Test

To enable or disable the scissor test, call `glEnable` or `glDisable` respectively, and specify `GL_SCISSOR_TEST` for *cap*. Call the `glIsEnabled()` function and specify `GL_SCISSOR_TEST` for *cap* to get the current setting. The scissor test is disabled by default. When it is disabled, fragments are not rejected.

#### Specifying the Scissor Box

Use the `glScissor()` function to specify the range (scissor box) through which fragments are allowed to pass.

**Code 10-6. Definition of the glScissor Function**

```
void glScissor(GLint x, GLint y, GLsizei width, GLsizei height);
```

*x* and *y* are coordinates that specify the starting point (lower-left corner) of the scissor box in the window coordinates. *width* and *height* specify the width and height of the scissor box. A `GL_INVALID_VALUE` error is generated if *width* or *height* is 0. Because there are no default values, always specify the scissor box when the scissor test is enabled.

The scissor box includes the fragment at the starting coordinates, but not the fragment with an x-coordinate of (*x* + *width*) or a y-coordinate of (*y* + *height*).

# 10.5. Rasterization

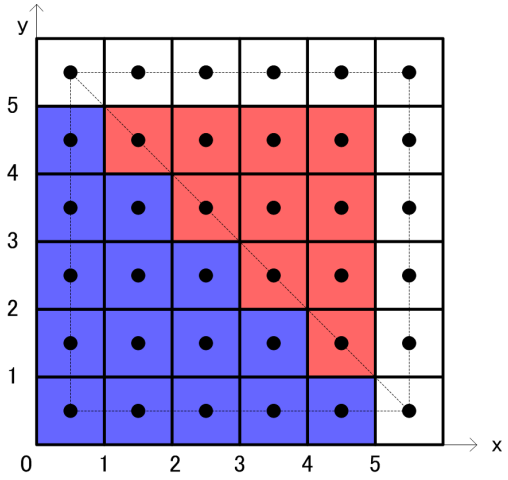
Polygon rasterization (fragment generation) by the PICA graphics core adheres to the following rules.

- The center coordinates of the pixel (x + 0.5, y + 0.5; x and y are integers) must lie inside the polygon.

- According to the lower-left rule, a fragment is generated if the lower or left edge of a polygon passes through the center coordinate of a pixel, but no fragment is generated if the upper or right edge passes through the center coordinate.
  **Note:** This assumes that the x-axis is negative toward the left and the y-axis is negative toward the lower edge.

Figure 10-2 shows how the lower-left rule is applied based on the rasterization results of two polygons. The two polygons are the one defined by the three vertices (5.5, 0.5), (5.5, 5.5), and (0.5, 5.5), and the one defined by the three vertices (5.5, 0.5), (0.5, 0.5), and (0.5, 5.5). The former is shown in red, while the latter is shown in blue.

**Figure 10-3. Example Demonstrating Rasterization Rules (the Lower-Left Rule)**




The rasterization rules are applied as follows.

On the boundary of the two polygons—because the left edge of the red polygon passes through the center of the pixels, these pixels are colored red.
Because the blue polygon passes through the center of pixels with a center coordinate x value of 0.5, these pixels are colored blue.
Because the right side of the red polygon passes through the center of the pixels with a center coordinate x value of 5.5, these pixels are not colored.
Because the bottom of the blue polygon passes through the center of the pixels with a center coordinate y value of 0.5, these pixels are colored blue.
And, because the top of the red polygon passes through the center of the pixels with a center coordinate y value of 5.5, these pixels are not colored.

CONFIDENTIAL

# 11. Texture Processing

The 3DS system can perform texture processing operations equivalent to those in OpenGL ES 2.0, but there are several CTR-specific restrictions.

## 11.1. Texture Unit

Four texture units (TEXTURE0 through TEXTURE3) are installed on the 3DS system, but each one can handle different types of textures. A rasterizer can independently output up to three sets of texture coordinates to the texture units. **To output texture coordinates to all four texture units, TEXTURE2 or TEXTURE3 must share the same texture coordinates with other texture units.**

**Table 11-1. Textures Supported by the Texture Units**

| Texture Unit | 2D Textures | Cube Map Textures | Shadow Textures | Projection Textures | Procedural Textures |
|---|---|---|---|---|---|
| TEXTURE0 | ✓ | ✓ | ✓ | ✓ | |
| TEXTURE1 | ✓ | | | | |
| TEXTURE2 | ✓ | | | | |
| TEXTURE3 | | | | | ✓ |

One- and three-dimensional textures are not supported. Cube map textures, shadow textures, projection textures, and other textures that require the w component can only be processed by TEXTURE0. TEXTURE3 is a unit used exclusively for procedural textures. GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS defines the number of installed texture units to be 4.

## 11.1.1. Texture Coordinate Input

Only texture unit 0 can accept the w component as input; the other texture units only allow two input components, u and v. A projection texture is simply a 2D texture with its w component enabled, but care must be taken with the coordinates output by the vertex shader because the generated UV coordinates are divided by the w component.

To send texture coordinates from the vertex shader, map output registers to the attribute names texture0, texture0w, texture1, and texture2. texture0w must be output for cube map textures, shadow textures, projection textures, and anything else that requires the w component. If it is not output, texture coordinate 0's output is undefined. texture0w output is ignored for textures that do not require the w component.

**Table 11-2. Attribute Names and Corresponding Texture Coordinates**

| Attribute Name | Attributes Sent From the Vertex Shader |
|---|---|
| texture0 | UV components of texture coordinate 0. |
| texture0w | W component of texture coordinate 0. |
| texture1 | UV components of texture coordinate 1. |
| texture2 | UV components of texture coordinate 2. |

### 11.1.1.1. Texture Coordinate Precision

Within a texture unit, texture coordinates are represented as 16-bit values combining an integer and decimal component. The number of decimal bits decreases as the absolute value of the integer component increases.

The accuracy of texture sampling depends on the decimal bit precision. A texture can be sampled, optimally, if there are enough decimal bits to represent its width and height in texels. Allocate another six decimal bits for bilinear filtering.

## 11.1.2. How to Use

To enable or disable a texture unit in OpenGL, specify an argument such as TEXTURE_2D to parameters `glEnable()` and `glDisable()`. On the 3DS system, however, set the reserved uniform `dmp_Texture[i].samplerType` (where *i* is the texture unit number). Behavior is undefined if you call `glEnable()` or `glDisable()` using the GL_TEXTURE_2D argument.

To disable a texture unit, call `glUniform1i`, and pass GL_FALSE to set a reserved uniform value. Each texture unit is enabled by different reserved uniform settings.

**Table 11-3. Settings for the Reserved Uniforms dmp_Texture[i].samplerType**

| Reserved Uniform | Value to Set | Supported Textures |
|---|---|---|
| `dmp_Texture[0].samplerType` | GL_FALSE | Disabled (default). |
| | GL_TEXTURE_2D | 2D textures |
| | GL_TEXTURE_CUBE_MAP | Cube-map texture. |
| | GL_TEXTURE_SHADOW_2D_DMP | Shadow textures. |
| | GL_TEXTURE_SHADOW_CUBE_DMP | Cube map shadow textures. |
| | GL_TEXTURE_PROJECTION_DMP | Projection textures |
| `dmp_Texture[1].samplerType` | GL_FALSE | Disabled (default). |
| | GL_TEXTURE_2D | 2D textures |
| `dmp_Texture[2].samplerType` | GL_FALSE | Disabled (default). |
| | GL_TEXTURE_2D | 2D textures |
| `dmp_Texture[3].samplerType` | GL_FALSE | Disabled (default). |
| | GL_TEXTURE_PROCEDURAL_DMP | Procedural textures |

Texture units 0 and 1 both have fixed texture coordinate input: texture coordinates 0 and 1, respectively. Texture units 2 and 3 must have their texture coordinate input specified by the reserved uniform `dmp_Texture[i].texcoord` (where *i* is the texture unit number 2 or 3). Each texture unit allows different reserved uniform settings. When all four texture units are used, texture unit 2 or 3 must share its input texture coordinates with another texture unit.

Color values input to the texture combiner are undefined when the texture combiner accesses a disabled texture unit.

**Table 11-4.Settings for the Reserved Uniforms dmp_Texture[i].texcoord**

| Reserved Uniform | Value to Set | Texture Coordinate Used as Input |
|---|---|---|
| `dmp_Texture[2].texcoord` | GL_TEXTURE1 | Texture coordinate 1 |
| | GL_TEXTURE2 | Texture coordinate 2 (default). |
| `dmp_Texture[3].texcoord` | GL_TEXTURE0 | Texture coordinate 0 (default). |
| | GL_TEXTURE1 | Texture coordinate 1 |
| | GL_TEXTURE2 | Texture coordinate 2 |

Unless they are configured using reserved uniforms, texture unit settings apply to the texture unit specified by the `glActiveTexture()` function.

**Code 11-1. Definition of the glActiveTexture Function**

```
void glActiveTexture(GLenum texture);
```

For *texture* you can specify GL_TEXTURE0, GL_TEXTURE1, or GL_TEXTURE2. A GL_INVALID_ENUM error is generated when some other value, or GL_TEXTURE3 (texture unit 3), is specified.

Reserved uniforms are used for all procedural texture settings.

## 11.1.3. Specifying Textures to Use

To specify the texture to use with a texture unit, first specify the texture unit using the glActiveTexture() function, and then specify the texture object using the glBindTexture() function.

If you want to use a different texture for each texture unit, call the glActiveTexture() and glBindTexture() functions, as shown in the following sample code.

**Code 11-2. Specifying a Texture for Each Texture Unit**

```
// Texture Unit0
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, imageTexID);
// Texture Unit1
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, bumpTexID);
```

## 11.1.4. Texture Parameters

Use the glTexParameter∗() functions to add parameters to a texture, such as the texture wrapping mode and filters.

**Code 11-3. Definition of the glTexParameter* Functions**

```
void glTexParameterf(GLenum target, GLenum pname, GLfloat param);
void glTexParameterfv(GLenum target, GLenum pname, const GLfloat* params);
void glTexParameteri(GLenum target, GLenum pname, GLint param);
void glTexParameteriv(GLenum target, GLenum pname, const GLint* params);
```

The glTexParameteri() function adds parameters passed in as integers and the glTexParameterf() function adds parameters passed in as floating-point numbers. Functions with names that end in v are used to add parameters that must be passed as vectors (arrays).

For *target* specify the same value specified in the glTexImage2D() function (Table 7-2). *pname* is the name of the parameter to add and *param* is the parameter value. Unique parameters have been added for 3DS.

The 3DS system gets up to eight texels for a single fragment when filtering.

**Table 11-5. Parameters Added for pname**

| pname | int / float / vector | Added Parameter |
|---|---|---|
| GL_TEXTURE_WRAP_S | int | Wrapping mode in the S direction (Table 11-6). |
| GL_TEXTURE_WRAP_T | int | Wrapping mode in the S direction (Table 11-6). |
| GL_TEXTURE_MIN_FILTER | int | Minification filter (Table 11-7). |
| GL_TEXTURE_MAG_FILTER | int | Magnification filter (Table 11-8). |

| | | |
|---|---|---|
| GL_TEXTURE_BORDER_COLOR | vec4(float) | The border color to use when the wrapping mode is GL_CLAMP_TO_BORDER (each component is between 0.0 and 1.0). |
| GL_TEXTURE_LOD_BIAS | float | LOD bias (−16.0 through 16.0. The default is 0.0.) Default value: 0.0 |
| GL_TEXTURE_MIN_LOD | int | Minimum LOD (−1000 by default.) |
| GL_GENERATE_MIPMAP | int(bool) | Automatic mipmap texture generation (GL_FALSE by default). For more information, see 11.1.4.1. Automatic Generation of Mipmap Textures. |

The following values specify the wrapping mode in the S and T directions.

**Table 11-6. Specifying the Wrapping Mode in the S and T Directions**

| param | Description |
|---|---|
| GL_REPEAT | Repeat (default). |
| GL_MIRRORED_REPEAT | Flip and repeat. |
| GL_CLAMP_TO_EDGE | Use the color at the edge of the texture image for texture coordinates that are not in the range from 0.0 through 1.0. |
| GL_CLAMP_TO_BORDER | Use the border color for texture coordinates that are not in the range from 0.0 through 1.0. |

The following values specify the filters to use when rendering texture images that have been scaled down.

**Table 11-7. Specifying Minification Filters**

| param | Description |
|---|---|
| GL_NEAREST | Use the color of the nearest texel (default). |
| GL_LINEAR | Use bilinear sampling (the average of four samples) to determine the color. |
| GL_NEAREST_MIPMAP_NEAREST | Select the nearest mipmap texture, and then use the color of the nearest texel. |
| GL_NEAREST_MIPMAP_LINEAR | Select two mipmap texture levels, and then interpolate between the nearest colors at each level. |
| GL_LINEAR_MIPMAP_NEAREST | Select the nearest mipmap texture, and then use bilinear sampling to determine the color. |
| GL_LINEAR_MIPMAP_LINEAR | Select two mipmap texture levels, and then interpolate between the bilinearly sampled colors at each level. |

The following values specify the filters to use when rendering texture images that have been scaled up.

**Table 11-8. Specifying Magnification Filters**

| param | Description |
|---|---|
| GL_NEAREST | Use the color of the nearest texel (default). |
| GL_LINEAR | Use bilinear sampling (the average of four samples) to determine the color. |

## 11.1.4.1. Automatic Generation of Mipmap Textures

When the texture parameter GL_GENERATE_MIPMAP is GL_TRUE and a value of −2 or less is passed to *level* in the glTexImage2D(), glCopyTexImage2D(), or glCopyTexSubImage2D() function, mipmap textures are automatically generated for all but the lowest mipmap level. However, texture formats (combinations of the *format* and *type* parameters) apply the following restrictions to the minimum width and height of mipmap textures that are automatically generated.

**Table 11-9. Minimum Width and Height of Automatically Generated Mipmap Textures**

| Format | format | type | Minimum Width and Height |
|---|---|---|---|
| RGBA4 | GL_RGBA<br>GL_RGBA_NATIVE_DMP | GL_UNSIGNED_SHORT_4_4_4_4 | 64 |
| RGBA5551 | GL_RGBA<br>GL_RGBA_NATIVE_DMP | GL_UNSIGNED_SHORT_5_5_5_1 | 64 |
| RGBA8 | GL_RGBA<br>GL_RGBA_NATIVE_DMP | GL_UNSIGNED_BYTE | 32 |
| RGB565 | GL_RGB<br>GL_RGB_NATIVE_DMP | GL_UNSIGNED_SHORT_5_6_5 | 64 |
| RGB8 | GL_RGB<br>GL_RGB_NATIVE_DMP | GL_UNSIGNED_BYTE | 32 |

Even if two textures have the same width and height, the range of values that can be specified for *level* differ, if the texture formats have different minimum values. A GL_INVALID_OPERATION error is generated when you specify the automatic generation of a mipmap texture smaller than the minimum size. For a texture that is 128×128 texels, for example, you can specify a *level* of −2 or −3 when the format is RGB8, but you can only specify −2 for *level* when the format is RGB565.

When enabled, automatically generated mipmap textures take priority. The mipmap texture data loaded with a texture image is ignored.

With automatic generation enabled, *level* for the glCopyTexImage2D() and glCopyTexSubImage2D() functions must have the same value as was passed to *level* for the glTexImage2D() function when the texture was loaded. A GL_INVALID_OPERATION error is generated if these values differ. However, even if a nonzero value is passed to *level*, data is copied to the texture with the lowest level, and the mipmap texture is not replaced.

With automatic generation disabled, *level* for the glCopyTexImage2D() and glCopyTexSubImage2D() functions must be 0, regardless of the value that was passed to *level* for the glTexImage2D() function when the texture was loaded. A GL_INVALID_OPERATION error is generated if a nonzero value is specified.

## 11.1.4.2. Cautions for Specifying GL_NEAREST as the Filter

If you set the GL_TEXTURE_MIN_FILTER and GL_TEXTURE_MAG_FILTER texture parameters to GL_NEAREST and use a texture image with vertical or horizontal lines (including color boundaries that form straight lines), when the texture is applied to polygons that place those stripes horizontally or perpendicular to the screen's scan lines, the straight lines may appear as broken lines.

This phenomenon is caused by the computational precision of texture coordinate interpolation within polygons. Consider a row of fragments that is horizontal or perpendicular to the scan lines. If each fragment samples texels in a single vertical or horizontal line, the straight lines on the texture image are rendered unchanged on the polygons. If each of these fragments sample around the boundary between two adjacent rows of texels, however, each fragment will sample a

different texel row due to texture coordinate errors. As a result, the straight line on the texture image is rendered as a broken line on the polygons. You can work around this by adjusting the texture coordinates and rendering area so that each fragment samples the texel centers, just as when a rectangular polygon with texture coordinates of 0 and 1 at its edges is rendered using the same size as a texture.

## 11.1.4.3. Cautions for Specifying GL_XXX_MIPMAP_LINEAR as a Minification Filter

Trilinear filtering is enabled when GL_XXX_MIPMAP_LINEAR is set for the texture parameter GL_TEXTURE_MIN_FILTER. This interpolates colors from two mipmap texture levels and then renders with the resulting color. However, this interpolation is subject to errors caused by computational precision. For example, even if two colors with the same component values are processed, a different color could be rendered due to these minor errors.

Colors are interpolated when, as a result of LOD calculations, they must be obtained from two mipmap texture levels. Colors are not interpolated when they are obtained from only one mipmap texture level. A texture is slightly darker where colors are interpolated than where colors are not. This causes differences in hue to be rendered as edges along mipmap level boundaries.

You can mitigate this effect by using fixed component values in a texel color. For example, a texture in the GL_RGB format has a fixed alpha component of 1.0, but even this fixed component value decreases slightly for texels that have been interpolated by trilinear filtering. Nonetheless, this component retains its value of 1.0 in texels that have not been interpolated. To correct a texture color, multiply it by the change in alpha value caused by interpolation, and then add this product to the texture color.

Texture combiners can make this correction, as shown in the following sample code. The following code shows an example of the setting.

**Code 11-4. Compensating for Trilinear Filtering**

```
glUniform1i(glGetUniformLocation(program, "dmp_TexEnv[0].combineRgb"),
        GL_MULT_ADD_DMP);
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[0].operandRgb"),
        GL_SRC_COLOR, GL_ONE_MINUS_SRC_ALPHA, GL_SRC_COLOR);
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[0].srcRgb"),
        GL_TEXTURE0, GL_TEXTURE0, GL_TEXTURE0);
```

Wherever colors are not interpolated, these settings result in a product of 0 and output the original texture color. Wherever colors are interpolated, these settings multiply the texture color and the difference in the alpha component, add this product to the texture color, and finally output the corrected color. You can only apply these settings to the first texture combiner (combiner 0) because they specify a texture color for every input source.

If you are using a texture format without fixed component values, such as GL_RGBA, prepare a separate texture with fixed component values and use that as a multitexture to perform the correction. This texture must have the same size, number of mipmap levels, and UV input values as the original texture. We recommend using an ETC1-compressed texture for data size and cache efficiency reasons.

## 11.1.4.4. Getting Texture Level (Mipmap) Parameters

You can use the following functions to get parameters for each mipmap level of textures bound to

texture units that are currently active. However, you cannot get information about procedural textures because the texture unit for procedural textures (`GL_TEXTURE3`) cannot be specified to the `glActiveTexture()` function.

**Code 11-5. Getting Texture Level Parameters**

```
void glGetTexLevelParameterfv(GLenum target, GLint level, GLenum pname,
                              GLfloat *params);
void glGetTexLevelParameteriv(GLenum target, GLint level, GLenum pname,
                              GLint *params);
```

Both functions get the same values, even though they are saved as different types.

Specify the type of texture in *`target`*. You can specify the following values.

**Table 11-10. Specifying the Types of Textures to Get Texture Level Parameters For**

| target Value | Type of Texture |
|---|---|
| `GL_TEXTURE_2D` | 2D texture (including shadow and gas textures). |
| `GL_TEXTURE_CUBE_MAP_POSITIVE_X` | Cube-map texture (in the positive X direction). |
| `GL_TEXTURE_CUBE_MAP_NEGATIVE_X` | Cube-map texture (in the negative X direction). |
| `GL_TEXTURE_CUBE_MAP_POSITIVE_Y` | Cube-map texture (in the positive Y direction). |
| `GL_TEXTURE_CUBE_MAP_NEGATIVE_Y` | Cube-map texture (in the negative Y direction). |
| `GL_TEXTURE_CUBE_MAP_POSITIVE_Z` | Cube-map texture (in the positive Z direction). |
| `GL_TEXTURE_CUBE_MAP_NEGATIVE_Z` | Cube-map texture (in the negative Z direction). |

Specify the mipmap level to get in *`level`*. When *`level`* is `0`, you get the parameter for the texture at the lowest mipmap level (the largest texture). You can get the next level and the one after that by specifying 1 and 2, respectively, for level.

Specify the type of parameter to get in *`pname`*. The following table shows how the value of *`pname`* corresponds to the parameter stored in *`params`*.

**Table 11-11. Specifying Texture Level Parameters to Get**

| pname Value | Parameters |
|---|---|
| `GL_TEXTURE_WIDTH` | Texture width (in texels). |
| `GL_TEXTURE_HEIGHT` | Texture height (in texels). |
| `GL_TEXTURE_DEPTH` | Unsupported; has a fixed value of 0. |
| `GL_TEXTURE_INTERNAL_FORMAT` | A texture's internal format. |
| `GL_TEXTURE_BORDER` | Unsupported; has a fixed value of 0. |
| `GL_TEXTURE_RED_SIZE` | Number of bits in the red component (per texel). |
| `GL_TEXTURE_GREEN_SIZE` | Number of bits in the green component (per texel). |
| `GL_TEXTURE_BLUE_SIZE` | Number of bits in the blue component (per texel). |
| `GL_TEXTURE_ALPHA_SIZE` | Number of bits in the alpha component (per texel). |
| `GL_TEXTURE_LUMINANCE_SIZE` | Number of bits in the luminance component (per texel). |
| `GL_TEXTURE_INTENSITY_SIZE` | Number of bits in the intensity component (per texel). Only for shadow textures. |
| `GL_TEXTURE_DEPTH_SIZE` | Number of bits in the depth component (per texel). Only for shadow textures. |
| `GL_TEXTURE_DENSITY1_SIZE_DMP` | Number of bits (per texel) of density information (density value 1) that does not account for intersections. Only for gas textures. |

| | |
|---|---|
| GL_TEXTURE_DENSITY2_SIZE_DMP | Number of bits (per texel) of density information (density value 2) that does account for intersections. Only for gas textures. |
| GL_TEXTURE_COMPRESSED | Whether a texture is compressed.<br>GL_TRUE: Compressed texture<br>GL_FALSE: Uncompressed texture |
| GL_TEXTURE_COMPRESSED_IMAGE_SIZE | Number of bytes in the texture at the mipmap level specified by *level*. Only for compressed textures. Specifying an uncompressed texture results in a GL_INVALID_OPERATION error. |

The following table shows how the number of bits in each component for a single texel corresponds to the internal format of textures obtained when GL_TEXTURE_INTERNAL_FORMAT is specified for *pname*.

**Table 11-12. Internal Formats and the Number of Bits in Components Making Up Each Texel**

| Internal Format | Red | Green | Blue | Alpha | Luminance | Intensity | Depth | Density 1 |
|---|---|---|---|---|---|---|---|---|
| GL_RGBA4 | 4 | 4 | 4 | 4 | | | | |
| GL_RGB5_A1 | 5 | 5 | 5 | 1 | | | | |
| GL_RGBA | 8 | 8 | 8 | 8 | | | | |
| GL_RGB565 | 5 | 6 | 5 | | | | | |
| GL_RGB | 8 | 8 | 8 | | | | | |
| GL_ALPHA | | | | 8 | | | | |
| GL_ALPHA4_EXT | | | | 4 | | | | |
| GL_LUMINANCE | | | | | 8 | | | |
| GL_LUMINANCE4_ EXT | | | | | 4 | | | |
| GL_LUMINANCE_ALPHA | | | | 8 | 8 | | | |
| GL_LUMINANCE4_ALPHA4_EXT | | | | 4 | 4 | | | |
| GL_SHADOW_DMP | | | | | | 8 | 24 | |
| GL_GAS_DMP | | | | | | | | 16 |
| GL_HILO8_DMP | 8 | 8 | | | | | | |
| GL_ETC1_RGB8_NATIVE_DMP | 8 | 8 | 8 | | | | | |
| GL_ETC1_ALPHA_RGB8_A4_NATIVE_DMP | 8 | 8 | 8 | 4 | | | | |

A GL_INVALID_ENUM error is generated when an invalid value is specified for *target* and *pname*. A GL_INVALID_VALUE error is generated when the mipmap level specified for *level* has not been loaded.

## 11.1.5. Effects of Texture Settings on Performance

Graphics performance is affected by texture format and size, and by various settings. The following is a list of common tendencies.

- Compressed textures are processed the fastest, followed by formats that use a small number of bytes per texel.
- Processing speed increases as the size decreases.
- Contention for memory access causes processing to slow as the number of textures used

simultaneously increases.

- The following pairs of minification filter settings are each processed at the same speed:GL_NEAREST and GL_LINEARGL_NEAREST_MIPMAP_NEAREST and GL_LINEAR_MIPMAP_NEARESTGL_NEAREST_MIPMAP_LINEAR and GL_LINEAR_MIPMAP_LINEAR. However, because GL_NEAREST(_XXX) fetches 1 texel per pixel and GL_LINEAR(_XXX) fetches 4 texels per pixel, GL_NEAREST(_XXX) uses less memory.
- It is faster to apply scaled-down textures using mipmaps. Even when mipmaps are used, however, the processing load depends on the filter settings. GL_∗_MIPMAP_LINEAR may entail approximately twice the processing load of GL_∗_MIPMAP_NEAREST.
- Although the GL_NEAREST and GL_LINEAR magnification filters have nearly the same performance, GL_NEAREST is slightly faster in some cases.
- Gas and shadow textures cannot use mipmaps and are processed more slowly than normal textures.
- Shadow textures make use of special filters for shadows, so there is a processing load comparable to trilinear filtering (when GL_∗_MIPMAP_LINEAR is set as the minification filter for normal textures). The processing load is around twice that for a normal texture (excluding trilinear filtering).
- There are no differences caused by conditions for setting procedural textures. They are faster to process than normal 2D textures.
- When multiple textures are used, they are processed more quickly if they are all placed in VRAM-A or VRAM-B, rather than split between the two.
- Textures created to match the upward direction of the framebuffer can sometimes process faster than textures created to match the upward direction of the rendering results. This is because the direction in which fragments are generated matches the direction in which textures are loaded, improving the texture cache hit rate. Flipping a texture vertically has no effect on performance. Fragments are processed horizontally in 8×8-pixel units, whereas textures are loaded in 8×4-texel units. Note also that the short sides of the 3DS's LCD screen are used as the top and bottom.

## 11.1.6. Texture Cache

There is a 256-byte L1 texture cache and an 8-KB L2 texture cache. Within a cache, only compressed textures (in the ETC format) are handled unchanged. All other textures (including those in the alpha ETC format) are converted into a 32-bit format.

There is a separate L1 cache for each texture unit, but the L2 cache is shared by all texture units.

There is a 5-cycle penalty for missing the L1 cache and, instead, getting data from the L2 cache. There is an additional penalty of approximately 30 cycles for missing the L2 cache and instead getting data from VRAM. However, the hardware is implemented to prefetch texels to hide these delays.

Texture caches have a 4-way set-associative format. There are 16 cache lines. The L2 cache is 8 KB, with 512 bytes per cache line.

A cache line is the lower 4 bits of a 8x4 block address (a 4x4 texel unit address divided by 2) calculated from the texture coordinate values. In ETC1, this is the value of the [5:2] bit of the address of the block unit in which 2x2 of the 4x4 texel blocks are arranged. Cache thrashing occurs when the same cache line is continuously accessed.

# 11.2. Combiners

There are six (texture) combiners installed on the 3DS system. They can combine the primary and secondary colors for fragment lighting, in addition to the colors output by texture units, such as the texture color, vertex color, and constant color. If you have experience developing applications for the Nintendo GameCube and Wii, you can more easily understand this effect if you imagine it as combining color and alpha values by using the TEV.

OpenGL ES 1.1 uses TexEnv for combiner settings, but the 3DS system uses reserved uniforms. The following table shows the reserved uniforms that correspond to TexEnv parameters.

**Table 11-13. Reserved Uniforms Corresponding to TexEnv Parameters in OpenGL ES 1.1**

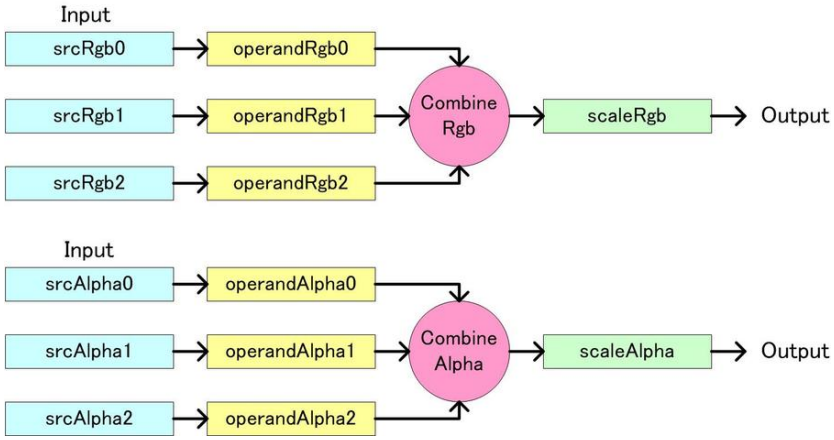
| TexEnv | Reserved Uniform | Setting |
|---|---|---|
| `COMBINE_RGB` | `dmp_TexEnv[i].combineRgb` | Color combiner function. |
| `COMBINE_ALPHA` | `dmp_TexEnv[i].combineAlpha` | Alpha combiner function. |
| `SRCn_RGB` | `dmp_TexEnv[i].srcRgb` | Color source. |
| `SRCn_ALPHA` | `dmp_TexEnv[i].srcAlpha` | Alpha source. |
| `OPERANDn_RGB` | `dmp_TexEnv[i].operandRgb` | Color operands. |
| `OPERANDn_ALPHA` | `dmp_TexEnv[i].operandAlpha` | Alpha operands. |
| `RGB_SCALE` | `dmp_TexEnv[i].scaleRgb` | Color scaling value. |
| `ALPHA_SCALE` | `dmp_TexEnv[i].scaleAlpha` | Alpha scaling value. |
| `TEXTURE_ENV_COLOR` | `dmp_TexEnv[i].constRgba` | Constant color (with an alpha component). |

Where *n* is the source (from `0` through 2) and *i* is the combiner number (from `0` through 5).

Each combiner processes its three source inputs as operands in its combiner function, multiplies the calculated result by a scaling value, and clamps the value between 0.0 and 1.0 before outputting it. Also, 0.0 to 1.0 are clamped before input is computed in the combiner, and the clamped value is a result of a rasterized absolute value for the vertex color (primary color).

Color operations are performed on all components (red, green, and blue) using a single setting, and alpha operations are performed on the alpha component using a separate setting. Color values input to the texture combiner are undefined when the texture combiner accesses a disabled texture unit.

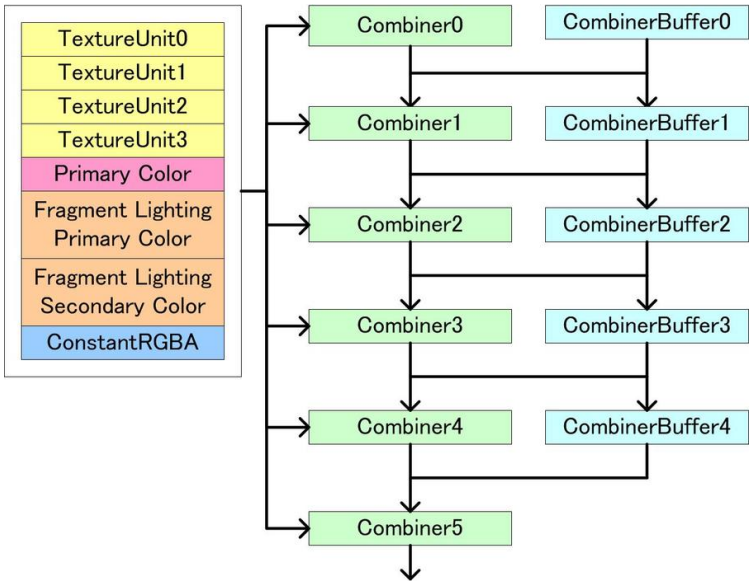
**Figure 11-1. Combiner Options**




A combiner accepts three input sources. Each input source must be one of the following types (a single type can be used for more than one input).

- Texture color output by a texture unit.
- Constant color.
- Primary color.
- Primary color for fragment lighting.
- Secondary color for fragment lighting.
- Output from the previous combiner stage (except for combiner 0).
- Output from the previous combiner buffer stage (except for combiner 0).

**Figure 11-2. Combiner Input Sources**




## 11.2.1. Reserved Uniforms for Combiner Functions

There are two reserved uniforms for combiner functions: `dmp_TexEnv[i].combineRgb` and `dmp_TexEnv[i].combineAlpha`. Use the `glUniform1i()` function to set a value at the reserved uniform location obtained by `glGetUniformLocation`.

The following values are used to set the reserved uniforms for combiner functions. The same values can be set for both `dmp_TexEnv[i].combineRgb` and `dmp_TexEnv[i].combineAlpha`.

If `GL_DOT3_RGBA` is set for the combiner, it must have the same combiner function (`GL_DOT3_RGBA`) for both the color (`combineRgb`) and alpha (`combineAlpha`) components.

**Table 11-14. Reserved Uniform Values That Can Be Set for Combiner Functions**

| Setting Value | Combiner Function |
|---|---|
| `GL_REPLACE` | Src0 (default) |
| `GL_MODULATE` | Src0 * Src1 |
| `GL_ADD` | Src0 + Src1 |
| `GL_ADD_SIGNED` | Src0 + Src1 - 0.5 |
| `GL_INTERPOLATE` | Src0 * Src2 + Src1 * (1 - Src2) |
| `GL_SUBTRACT` | Src0 - Src1 |
| `GL_DOT3_RGB` | 4 * ((Src0_Red - 0.5) * (Src1_Red - 0.5) + (Src0_Green - 0.5) * (Src1_Green - 0.5) + (Src0_Blue - 0.5) * (Src1_Blue - 0.5)) |
| | 4 * ((Src0_Red - 0.5) * (Src1_Red - 0.5) + (Src0_Green - 0.5) * (Src1_Green - |

| | |
|---|---|
| GL_DOT3_RGBA | 0.5) + (Src0_Blue - 0.5) * (Src1_Blue - 0.5)) |
| GL_ADD_MULT_DMP | (Src0 + Src1) * Src2 **Note:** The sum is clamped between 0.0 and 1.0 before it is multiplied. |
| GL_MULT_ADD_DMP | (Src0 * Src1) + Src2 |

## 11.2.2. Reserved Uniforms for the Input Sources

There are two reserved uniforms for the input sources: `dmp_TexEnv[i].srcRgb` and `dmp_TexEnv[i].srcAlpha`.

Use the `glUniform3i()` function to set values at the reserved uniform location obtained by the `glGetUniformLocation()` function. Source 0 is first, followed by source 1 and source 2, respectively.

The following values are used to set the reserved uniforms for the input sources. The same values can be set for `dmp_TexEnv[0].srcRgb` and `dmp_TexEnv[0].srcAlpha`, and for `dmp_TexEnv[i].srcRgb` and `dmp_TexEnv[i].srcAlpha`.

**Warning:** Every combiner except for combiner 0 must have `GL_CONSTANT`, `GL_PREVIOUS`, or `GL_PREVIOUS_BUFFER_DMP` specified as one of its three input sources.

**Table 11-15. Reserved Uniform Values That Can Be Set for Input Sources**

| Setting Value | Input Source |
|---|---|
| GL_TEXTURE0 | Texture color from texture unit 0. |
| GL_TEXTURE1 | Texture color from texture unit 1. |
| GL_TEXTURE2 | Texture color from texture unit 2. |
| GL_TEXTURE3 | Texture color from texture unit 3. |
| GL_CONSTANT | Constant color. (This is the default for combiner 0 and is set by `dmp_TexEnv[i].constRgba`.) |
| GL_PRIMARY_COLOR | Primary color (vertex color). |
| GL_PREVIOUS | Output from the previous combiner stage. (Cannot be set for combiner 0. Default for combiners other than 0.) |
| GL_PREVIOUS_BUFFER_DMP | Output from the previous combiner buffer stage. (Cannot be set for combiner 0.) |
| GL_FRAGMENT_PRIMARY_COLOR_DMP | Primary color for fragment lighting. |
| GL_FRAGMENT_SECONDARY_COLOR_DMP | Secondary color for fragment lighting. |

## 11.2.3. Reserved Uniforms for Operands

There are two reserved uniforms for operands: `dmp_TexEnv[i].operandRgb` and `dmp_TexEnv[i].operandAlpha`.

Use the `glUniform3i()` function to set values at the reserved uniform location obtained by the `glGetUniformLocation()` function. Source 0 is first, followed by source 1 and source 2, respectively.

The following values are used to set the reserved uniforms for operands.

**Table 11-16. Reserved Uniform Values That Can Be Set for Operands**

| Setting Value | Operands |
|---|---|
| GL_SRC_COLOR | Color. (Cannot be set for operandAlpha. operandAlpha is the default.) |
| GL_ONE_MINUS_SRC_COLOR | 1 - Color. (Cannot be set for operandAlpha.) |
| GL_SRC_ALPHA | Alpha. (operandAlpha is the default.) |
| GL_ONE_MINUS_SRC_ALPHA | 1 - Alpha |
| GL_SRC_R_DMP | Color_Red |
| GL_ONE_MINUS_SRC_R_DMP | 1 - Color_Red |
| GL_SRC_G_DMP | Color_Green |
| GL_ONE_MINUS_SRC_G_DMP | 1 - Color_Green |
| GL_SRC_B_DMP | Color_Blue |
| GL_ONE_MINUS_SRC_B_DMP | 1 - Color_Blue |

## 11.2.4. Reserved Uniforms for Scaling Values

There are two reserved uniforms for scaling values: dmp_TexEnv[*i*].scaleRgb and dmp_TexEnv[*i*].scaleAlpha.

Use the glUniform1f() function to set a value at the reserved uniform location obtained by the glGetUniformLocation() function

The following values are used to set the reserved uniforms for scaling.

**Table 11-17. Reserved Uniform Values That Can Be Set for Scaling**

| Setting Value | Scaling |
|---|---|
| 1.0 | Unchanged combiner output (default). |
| 2.0 | Double the combiner output (clamp between 0.0 and 1.0). |
| 4.0 | Quadruple the combiner output (clamp between 0.0 and 1.0). |

## 11.2.5. Reserved Uniforms for Constant Colors

There is one uniform for constant colors: dmp_TexEnv[*i*].constRgba. Use the glUniform4f() function to set values at the reserved uniform location obtained by the glGetUniformLocation() function. The first value is R, followed by G, B, and A, respectively.

By default, 0.0 is set for the R, G, B, and A values in the reserved uniforms for constant colors.

## 11.2.6. Sample Combiner Settings

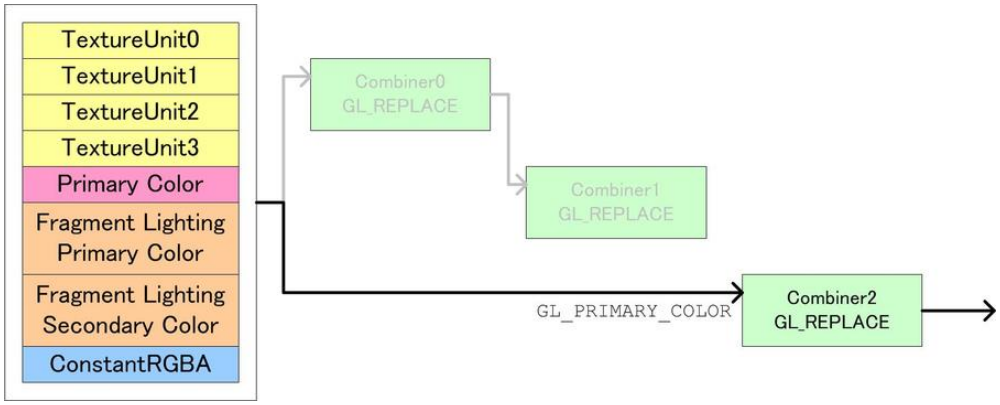
In the following example, combiner 2 references the primary color and outputs (renders) it unchanged.

To output only the primary color from the combiners without being affected by anything else, combiner 2's input source 0 is set to the primary color (GL_PRIMARY_COLOR), operand 0 is set to the unchanged input source color (GL_SRC_COLOR), the combiner function is configured to output

input source 0 unchanged (GL_REPLACE), and the scale is set to 1.0. With these settings, the combiner outputs only the primary color and combiners 0 and 1 do not affect the output results.

This is illustrated by the following connection diagram.

**Figure 11-3. Sample Combiner Settings 1**



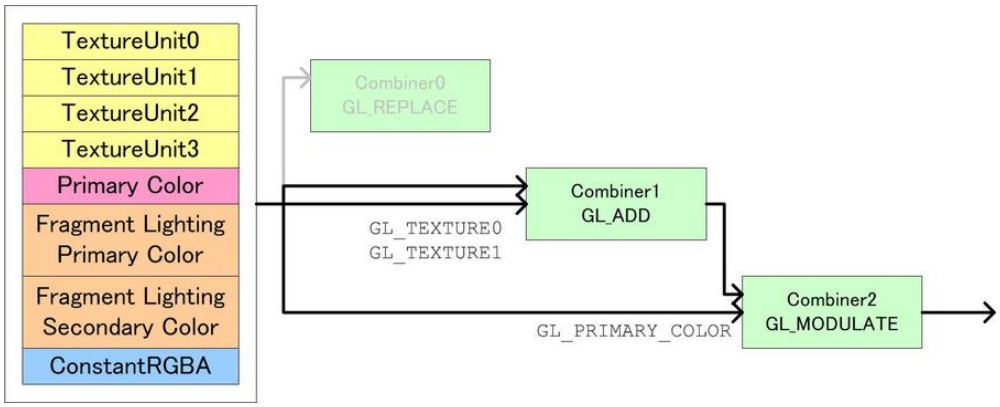

The following sample code shows how to make these settings in a program.

**Code 11-6. Code for Sample Combiner Settings 1**

```
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[2].srcRgb"),
        GL_PRIMARY_COLOR, GL_PREVIOUS, GL_PREVIOUS);
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[2].srcAlpha"),
        GL_PRIMARY_COLOR, GL_PREVIOUS, GL_PREVIOUS);
glUniform1i(glGetUniformLocation(program, "dmp_TexEnv[2].combineRgb"),
        GL_REPLACE);
glUniform1i(glGetUniformLocation(program, "dmp_TexEnv[2].combineAlpha"),
        GL_REPLACE);
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[2].operandRgb"),
        GL_SRC_COLOR, GL_SRC_COLOR, GL_SRC_COLOR);
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[2].operandAlpha"),
        GL_SRC_ALPHA, GL_SRC_ALPHA, GL_SRC_ALPHA);
glUniform1f(glGetUniformLocation(program, "dmp_TexEnv[2].scaleRgb"), 1.0);
glUniform1f(glGetUniformLocation(program, "dmp_TexEnv[2].scaleAlpha"), 1.0);
```

In a more complex example, combiner 1 is configured to add output from texture 0 and texture 1, and combiner 2 is configured to multiply the result by the primary color.

**Figure 11-4. Sample Combiner Settings 2**




**Code 11-7. Code for Sample Combiner Settings 2**

```
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[1].srcRgb"),
        GL_TEXTURE0, GL_TEXTURE1, GL_PREVIOUS);
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[1].srcAlpha"),
        GL_TEXTURE0, GL_TEXTURE1, GL_PREVIOUS);
```

```
glUniform1i(glGetUniformLocation(program, "dmp_TexEnv[1].combineRgb"),
        GL_ADD);
glUniform1i(glGetUniformLocation(program, "dmp_TexEnv[1].combineAlpha"),
        GL_ADD);
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[1].operandRgb"),
        GL_SRC_COLOR, GL_SRC_COLOR, GL_SRC_COLOR);
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[1].operandAlpha"),
        GL_SRC_ALPHA, GL_SRC_ALPHA, GL_SRC_ALPHA);
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[2].srcRgb"),
        GL_PREVIOUS, GL_PRIMARY_COLOR, GL_PREVIOUS);
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[2].srcAlpha"),
        GL_PREVIOUS, GL_PRIMARY_COLOR, GL_PREVIOUS);
glUniform1i(glGetUniformLocation(program, "dmp_TexEnv[2].combineRgb"),
        GL_MODULATE);
glUniform1i(glGetUniformLocation(program, "dmp_TexEnv[2].combineAlpha"),
        GL_MODULATE);
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[2].operandRgb"),
        GL_SRC_COLOR, GL_SRC_COLOR, GL_SRC_COLOR);
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[2].operandAlpha"),
        GL_SRC_ALPHA, GL_SRC_ALPHA, GL_SRC_ALPHA);
```

## 11.3. Combiner Buffers

In the CTR system, each combiner, except for the last one (combiner 5), has a combiner buffer configured in parallel to it. A combiner buffer can select the output of the previous combiner stage or combiner buffer stage as its input source. By preserving the output of a previous combiner buffer stage, a combiner buffer can allow a later combiner stage's input to come from the output of a combiner stage earlier than the previous one.

### 11.3.1. Reserved Uniforms for Combiner Buffers

Because combiner buffer 0 has no input, its initial value is a constant color set by the reserved uniform `dmp_TexEnv[0].bufferColor`. Use the `glUniform4f()` function to set the constant color at the reserved uniform location obtained by the `glGetUniformLocation()` function. The first value is R, followed by G, B, and A, respectively.

By default, 0.0 is set as the R, G, B, and A values of combiner buffer 0's constant color.

You can choose output from either the previous combiner stage or the previous combiner buffer stage as the input source to combiner buffers 1 through 4. Input from the color and alpha components can be selected separately through the reserved uniforms `dmp_TexEnv[`*i*`].bufferInput` (where *i* is 1 through 4). Use the `glUniform2i()` function to set a value at the reserved uniform location obtained by `glGetUniformLocation`. The color components are first, followed by the alpha component.

The following reserved uniform values are used to configure the combiner buffer input sources.

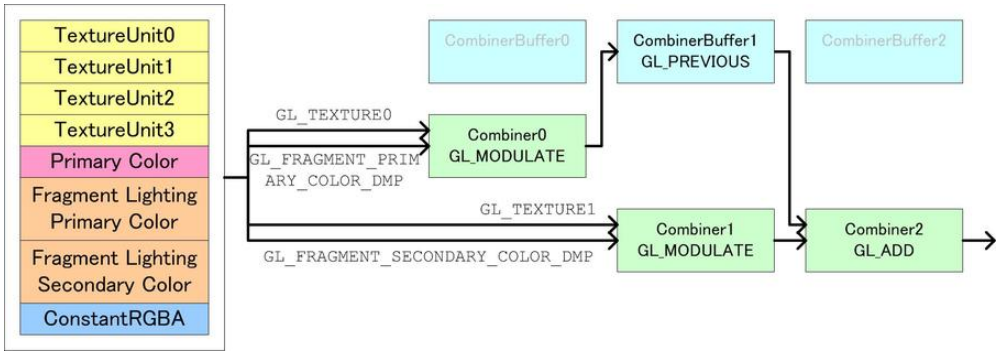
**Table 11-18. Reserved Uniform Values That Can Be Set for Combiner Buffer Input Sources**

| Setting Value | Input Source |
|---|---|
| `GL_PREVIOUS` | Output from the previous combiner stage (default). |
| `GL_PREVIOUS_BUFFER_DMP` | Output from the previous combiner buffer stage. |

### 11.3.2. Sample Combiner Buffer Settings

The following sample settings demonstrate how to connect combiner buffers so that texture 0 is multiplied with the primary color for fragment lighting, texture 1 is multiplied with the secondary color for fragment lighting, and then these results are added together and output.

**Figure 11-5. Sample Combiner Buffer Settings**




The following sample code shows how to make these settings in a program.

**Code 11-8. Sample Code for Setting Up the Combiner Buffers**

```
// Combiner 0
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[0].srcRgb"),
        GL_TEXTURE0, GL_FRAGMENT_PRIMARY_COLOR_DMP, GL_PREVIOUS);
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[0].srcAlpha"),
        GL_TEXTURE0, GL_PREVIOUS, GL_PREVIOUS);
glUniform1i(glGetUniformLocation(program, "dmp_TexEnv[0].combineRgb"),
        GL_MODULATE);
glUniform1i(glGetUniformLocation(program, "dmp_TexEnv[0].combineAlpha"),
        GL_REPLACE);
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[0].operandRgb"),
        GL_SRC_COLOR, GL_SRC_COLOR, GL_SRC_COLOR);
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[0].operandAlpha"),
        GL_SRC_ALPHA, GL_SRC_ALPHA, GL_SRC_ALPHA);
// CombinerBuffer 1
glUniform2i(glGetUniformLocation(program, "dmp_TexEnv[1].bufferInput"),
        GL_PREVIOUS, GL_PREVIOUS);
// Combiner 1
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[1].srcRgb"),
        GL_TEXTURE1, GL_FRAGMENT_SECONDARY_COLOR_DMP, GL_PREVIOUS);
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[1].srcAlpha"),
        GL_TEXTURE1, GL_PREVIOUS, GL_PREVIOUS);
glUniform1i(glGetUniformLocation(program, "dmp_TexEnv[1].combineRgb"),
        GL_MODULATE);
glUniform1i(glGetUniformLocation(program, "dmp_TexEnv[1].combineAlpha"),
        GL_REPLACE);
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[1].operandRgb"),
        GL_SRC_COLOR, GL_SRC_COLOR, GL_SRC_COLOR);
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[1].operandAlpha"),
        GL_SRC_ALPHA, GL_SRC_ALPHA, GL_SRC_ALPHA);
// Combiner 2
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[2].srcRgb"),
        GL_PREVIOUS_BUFFER_DMP, GL_PREVIOUS, GL_PREVIOUS);
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[2].srcAlpha"),
        GL_PREVIOUS_BUFFER_DMP, GL_PREVIOUS, GL_PREVIOUS);
glUniform1i(glGetUniformLocation(program, "dmp_TexEnv[2].combineRgb"),
        GL_ADD);
glUniform1i(glGetUniformLocation(program, "dmp_TexEnv[2].combineAlpha"),
        GL_REPLACE);
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[2].operandRgb"),
        GL_SRC_COLOR, GL_SRC_COLOR, GL_SRC_COLOR);
glUniform3i(glGetUniformLocation(program, "dmp_TexEnv[2].operandAlpha"),
        GL_SRC_ALPHA, GL_SRC_ALPHA, GL_SRC_ALPHA);
```

## 11.4. Procedural Textures

Procedural textures are unlike conventional textures, in that they determine texel colors procedurally rather than by referencing images. Procedural textures are most effective when used for perfectly regular patterns and patterns that are regular, but have some randomness. Because they do not access texture images, they avoid memory access conflicts and reduce data (content) sizes.

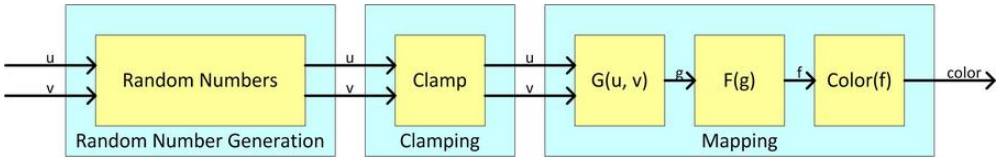
Texture unit 3 is exclusively used for procedural textures, and is also the only texture unit that can handle procedural textures. Even though procedural textures calculate texel colors, they are similar to normal textures, in that they still determine which texel colors correspond to UV texture coordinates.

As a part of the reserved fragment shaders, procedural textures have parameters that are configured through reserved uniforms by the `glUniform*()` functions.

## 11.4.1. Procedural Texture Unit

The procedural texture unit comprises three computational components. In order of process flow, these are random number generation, clamping, and mapping. Random number generation adds noise to UV texture coordinates; clamping determines wrapping and mirror symmetry for patterns; and mapping calculates texel colors from UV coordinates.

**Figure 11-6. Structure of the Procedural Texture Unit**




Input texture coordinates are processed as shown in the figure above. Use the following procedure to set parameters to get the desired image.

1. Enable Procedural Textures
2. Set shared RGBA mode or independent alpha mode.
   This corresponds to settings for G(u, v) and F(g).
3. Select a basic shape.
   This corresponds to the selection in G(u, v).
4. Colors are set in color lookup tables. This corresponds to settings for Color(f).
5. Set the relationship between the basic shape and color lookup table.
   This determines how the basic shape in step 3 corresponds to the color lookup table in step 4. This corresponds to settings for F(g).
6. Select random number parameters.
   If necessary, enable random numbers and determine the size of their effect. If not necessary, disable them. This corresponds to random number generation. The UV coordinates are made into absolute values and output to the clamp portion, regardless of whether they are enabled or disabled.
7. Configure wrapping and symmetry.
   This corresponds to clamp settings.

## 11.4.2. Enable Procedural Textures

To enable or disable texture unit 3, which is used by procedural textures, set a value for the `dmp_Texture[3].samplerType` reserved uniform. Note that using `glActiveTexture` to select a unit or `glEnable` to select a texture type results in an error.

Using `glUniform1i`, set the value to `GL_TEXTURE_PROCEDURAL_DMP` to enable texture unit 3 or to `GL_FALSE` to disable it. The only supported texture type is `GL_TEXTURE_PROCEDURAL_DMP`.

**Code 11-9. Enabling Procedural Textures**

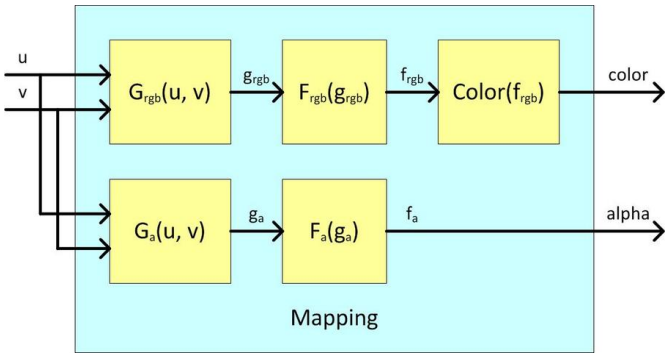
```
glUniform1i(
    glGetUniformLocation(s_PgID, "dmp_Texture[3].samplerType"),
    GL_TEXTURE_PROCEDURAL_DMP);
```

## 11.4.3. Setting Shared RGBA Mode or Independent Alpha Mode

Choose whether the alpha component is mapped using the same functions as the RGB components (shared RGBA mode) or using functions that are set separately for the alpha component (independent alpha mode). If independent alpha mode is selected, two G functions and two F functions must be set for mapping. Shared RGBA mode is easier to set if you just want to output an image to see what it looks like.

Using the `glUniform1i()` function, set the reserved uniform `dmp_Texture[3].ptAlphaSeparate` to `GL_FALSE` to select shared RGBA mode or to `GL_TRUE` to select independent alpha mode. Shared RGBA mode is the default.

**Figure 11-7. Mapping in Independent Alpha Mode**




## 11.4.4. Selecting a Basic Shape

A procedural texture's basic shape is determined by the G function during mapping.

The basic shapes in Table 11-19 use the following color assignments.

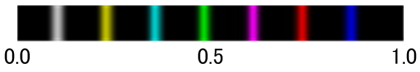
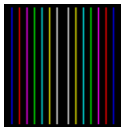
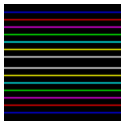
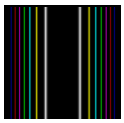
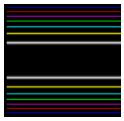
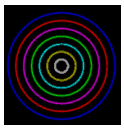
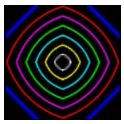



**Table 11-19. G Function Settings, Selected Functions, and Basic Shapes**

| Setting Value | Selected Function | Basic Shape |
|---|---|---|
| GL_PROCTEX_U_DMP (default) | u |  |
| | | |

| | | |
|---|---|---|
| GL_PROCTEX_V_DMP | v |  |
| GL_PROCTEX_U2_DMP | $u^2$ |  |
| GL_PROCTEX_V2_DMP | $v^2$ |  |
| GL_PROCTEX_ADD_DMP | (u + v) / 2 | |
| GL_PROCTEX_ADD2_DMP | $(u^2 + v^2) / 2$ | |
| GL_PROCTEX_ADDSQRT2_DMP | $sqrt(u^2 + v^2)$ |  |
| GL_PROCTEX_MIN_DMP | min(u, v) | |
| GL_PROCTEX_MAX_DMP | max(u, v) | |
| GL_PROCTEX_RMAX_DMP | $((u + v) / 2 + sqrt(u^2 + v^2)) / 2$ |  |

The basic shape is rendered using texture coordinates between `-1.0` and `1.0` (with `0.0` at the center) and GL_MIRRORED_REPEAT specified for wrapping.

The reserved uniforms used for selection with the G function are `dmp_Texture[3].ptRgbMap` for RGB components and `dmp_Texture[3].ptAlphaMap` for the alpha component. Use the `glUniform1i()` function to set each of these reserved uniforms.

`dmp_Texture[3].ptAlphaMap` settings are only valid in independent alpha mode. Choose the shape that is closest to the desired texture image.

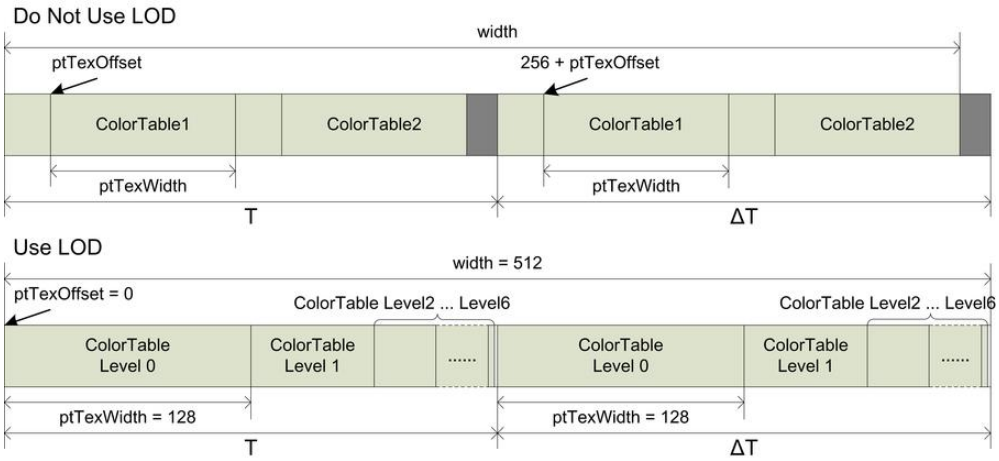
For example, choose GL_PROCTEX_U_DMP and GL_PROCTEX_V_DMP for wood grain or GL_PROCTEX_ADDSQRT2_DMP for the annual rings in a tree.

## 11.4.5. Color Lookup Table Settings

Color lookup tables are used to convert values calculated by the G and F functions into actual texel colors, and can set each of the RGBA components separately.

The content of the color lookup table depends on whether LOD is used.

**Figure 11-8. Color Lookup Table Differences Caused by the Use of LOD**




When LOD is not used, multiple color tables can be stored as partial arrays in a color lookup table. By changing the offset and table width, you can also render textures with different coloring from the same calculation results.

A color lookup table can hold up to 512 elements. You can store the color table in the first 256 elements and the differences between the color table values in the last 256 elements. Because the delta values must start at the 257$^{th}$ element, the number of color lookup table entries is defined as the sum of 256, the number of color lookup table elements that are actually referenced, and the starting offset to the color tables. The number of elements is calculated for the last color table.

Use the `glUniform1i()` function to set the color table width that is actually referenced (`dmp_Texture[3].ptTexWidth`). Set this value to a power of 2 that is no greater than 128. Use the `glUniform1i()` function to set the color table's starting offset (`dmp_Texture[3].ptTexOffset`) to an integer between 0 and 128.

When LOD is used, the color table width and offset must be 128 and 0 respectively. The level of detail determines which color table is actually referenced. The maximum number of elements in a color lookup table is fixed at 512.

**Table 11-20. Level of Detail and Corresponding Color Tables**

| Level of Detail | Starting Position | Width |
|---|---|---|
| 0 | 0 | 128 |
| 1 | 128 | 64 |
| 2 | 192 | 32 |
| 3 | 224 | 16 |
| 4 | 240 | 8 |
| 5 | 248 | 4 |
| 6 | 252 | 2 |

As explained in 7.7. Loading Lookup Tables, lookup tables are loaded from arrays by calls to the `glTexImage1D()` function. Prepare an array of floating-point numbers with as many elements as the color lookup table, storing values `0.0` through `1.0` for the color table elements (T) in the first half of the array and the differences between the first half's 256 elements (ΔT) in the last half of the array. The following equations calculate the elements and delta values, given *size* as the number of

elements in the color table, *offset* as the starting offset, $C_i$ as each element, and *func* as the conversion function.

$$C_{i+offset} = func\left(\frac{i}{size}\right) \qquad (0 \le i < size)$$

$$C_{i+offset+256} = C_{i+offset+1} - C_{i+offset} \qquad (0 \le i < size)$$

Set the last difference value to either `0.0` or the difference between the color table's last element and the convergence value.

You can use the following code to call the `glTexImage1D()` function and configure 512 as the maximum number of elements in a color lookup table, `data` as the array storing the color lookup table, and 0 as the number of the lookup table to set.

**Code 11-10. Loading a Color Lookup Table**

```
glTexImage1D(GL_LUT_TEXTURE0_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
             GL_LUMINANCEF_DMP, GL_FLOAT, data);
```

The color lookup table used by each RGBA component uses the `glUniform1i()` function to specify the lookup table number as the following reserved uniform. Note that the lookup table number specified here, `GL_LUT_TEXTURE`*`i`*`_DMP`, where *i* represents a number from 0 to 31, does not specify the name (ID) of the texture nor `GL_LUT_TEXTURE`*`i`*`_DMP` directly.

**Table 11-21. Reserved Uniforms That Specify Color Lookup Tables**

| Reserved Uniform | Value to Set |
|---|---|
| `dmp_Texture[3].ptSamplerR` | Specifies the lookup table number to use as the color lookup table for the red component. |
| `dmp_Texture[3].ptSamplerG` | Specifies the lookup table number to use as the color lookup table for the green component. |
| `dmp_Texture[3].ptSamplerB` | Specifies the lookup table number to use as the color lookup table for the blue component. |
| `dmp_Texture[3].ptSamplerA` | Specifies the lookup table number to use as the color lookup table for the alpha component. |

Each reserved uniform specifies a value between 0 and 31.

The value set for `dmp_Texture[3].ptSamplerA` is ignored in independent alpha mode.

You can apply the same minification filters to a procedural texture's color lookup tables as you can to a normal texture. Choose a value from the following table to set the reserved uniform `dmp_Texture[3].ptMinFilter` with the `glUniform1i()` function.

**Table 11-22. Color Lookup Table Filters**

| Value to Set | Filter to Apply |
|---|---|
| `GL_NEAREST` | Nearest in the UV directions without LOD. |
| `GL_LINEAR` | Linear in the UV directions without LOD (default). |
| `GL_NEAREST_MIPMAP_NEAREST` | Nearest in the UV directions with the nearest LOD. |
| `GL_NEAREST_MIPMAP_LINEAR` | Nearest in the UV directions with a linear LOD. |
| `GL_LINEAR_MIPMAP_NEAREST` | Linear in the UV directions with the nearest LOD. |
| `GL_LINEAR_MIPMAP_LINEAR` | Linear in the UV directions with a linear LOD. |

You can apply an LOD bias when referencing the color lookup table. Use the `glUniform1f()` function to set the reserved uniform `dmp_Texture[3].ptTexBias` to a value from 0.0 through 6.0. This is disabled with a value of 0.0, and has a default value of 0.5.

## 11.4.6. Setting the Relationship Between the Basic Shape and Color Lookup Table

The F function configures how the G function, which selects the basic shape, corresponds to the color lookup table, which sets the basic color. The F function uses a lookup table to map output from the G function (0.0 through 1.0) into lookup values for the color lookup table (0.0 through 1.0). The lookup table has 256 elements. A mapping table is stored in the first 128 elements and the differences between the mapping table values are stored in the last 128 elements. Because the mapping table configures the relationship between shapes and colors, by changing the mapping table you can render textures that have different appearances even though they use the same shape and color lookup table. By modifying the F function, a wide variety of outputs is possible. For example, you could use a simple F function to calculate results, such as $F(x)=x$ or $F(x)=x^2$, or you could have the F function use discontinuous values and operate like an index.

Like color lookup tables, the mapping table for the F function is loaded from an array into a lookup table by a call to the `glTexImage1D()` function. Prepare an array of floating-point numbers with as many elements (256) as the mapping table, storing the mapping table elements (0.0 through 1.0) in the first half of the array and the differences between those elements in the last half of the array. The following equations calculate the mapping table elements and delta values, assuming $F_i$ is a mapping table element and *func* is the conversion function.

$$F_i = func\left(\frac{i}{128.0}\right) \qquad (0 \le i \le 127)$$

$$F_{k+128} = func\left(\frac{k+1}{128.0}\right) - func\left(\frac{k}{128.0}\right) \quad (0 \le k \le 127)$$

You can use the following code to call the `glTexImage1D()` function and configure 256 as the number of mapping table elements, `data` as the array storing the mapping table, and 0 as the number of the lookup table to set.

**Code 11-11. Loading a Mapping Table**

```
glTexImage1D(GL_LUT_TEXTURE0_DMP, 0, GL_LUMINANCEF_DMP, 256, 0,
             GL_LUMINANCEF_DMP, GL_FLOAT, data);
```

The mapping table used as the F functions uses the `glUniform1i()` function to specify the lookup table number as the following reserved uniform. Note that the specified lookup table number `GL_LUT_TEXTURE`*i*`_DMP`, where *i* represents a number from 0 to 31, does not specify the name (ID) of the texture nor `GL_LUT_TEXTURE`*i*`_DMP` directly.

**Table 11-23. Reserved Uniforms That Specify Mapping Tables**

| Reserved Uniform | Value to Set |
|---|---|
| `dmp_Texture[3].ptSamplerRgbMap` | Specifies the lookup table number to use as the F function for RGB values. |
| `dmp_Texture[3].ptSamplerAlphaMap` | Specifies the lookup table number to use as the F function for alpha values. |

Each reserved uniform specifies a value between 0 and 31.
`dmp_Texture[3].ptSamplerAlphaMap` is only valid in independent alpha mode.

## 11.4.7. Selecting Random Number Parameters

As a random element in a procedural texture, noise can be added to the UV texture coordinates that are input to the G function. Noise affects the basic shape. When the G function is `GL_PROCTEX_U_DMP` and noise affects U texture coordinates, for example, it becomes possible to render wood grain with natural warping. Ordinarily, wood grain can only be rendered in a straight line.

To enable or disable noise, set a value in the reserved uniform `dmp_Texture[3].ptNoiseEnable`. Using `glUniform1i`, specify `GL_TRUE` to enable it, and `GL_FALSE` to disable it.

**Code 11-12. Enabling Noise**

```
glUniform1i(
    glGetUniformLocation(s_PgID, "dmp_Texture[3].ptNoiseEnable"), GL_TRUE);
```

The function that adds noise is a black box, but it can be controlled through three parameters from the application: the frequency (F), the phase (P), and the amplitude (A). The F parameter adjusts the speed of the fluctuations (noise) so that large values create jagged waves and small values create gentle undulations. The P parameter changes the starting location of the noise. When rendering a texture of the ocean surface, for example, you can represent changing waves by modifying only the P parameter. When increased, the A parameter magnifies the effect of the noise and further destroys the basic shape.

The three parameters F, P, and A can each be set separately for the U and V components.

**Table 11-24. Reserved Uniforms for Noise**

| Reserved Uniform | Value to Set |
|---|---|
| `dmp_Texture[3].ptNoiseU` | Specifies the F, P, and A parameters for the u-component. The range from -8.0 to 8.0 is clamped only for parameter A.<br>(F-parameter, P-parameter, A-parameter)<br>These are (0.0, 0.0, 0.0) by default. |
| `dmp_Texture[3].ptNoiseV` | Specifies the F, P, and A parameters for the v component. The range from -8.0 to 8.0 is clamped only for parameter A.<br>(F-parameter, P-parameter, A-parameter)<br>These are (0.0, 0.0, 0.0) by default. |

Apart from the noise parameters, you can control changes in the continuity of random numbers (called noise modulation) in the function that adds noise. Noise modulation (a continuous noise function) is specified by a lookup table called the noise modulation table. The noise function takes a noise modulation table and uses it to create natural noise values from the discrete values that arise from calculations alone. A suitable continuous noise function, such as $3x^2 - 2x^3$, generates values that change gradually when x is near `0.0` and `1.0`.

Like the color lookup table, the noise modulation table is loaded from an array by a call to the `glTexImage1D()` function. The noise function takes a noise modulation table and uses it to create natural noise values from the discrete values that arise from calculations alone. The following equations calculate the noise modulation table elements and delta values, assuming $N_i$ is a table element and *func* is the conversion function.

$$N_i = func\left(\frac{i}{128.0}\right) \qquad (0 \le i \le 127)$$

$$N_{k+128} = func\left(\frac{k+1}{128.0}\right) - func\left(\frac{k}{128.0}\right) \quad (0 \le k \le 127)$$

You can use the following code to call the `glTexImage1D()` function and configure 256 as the number of noise modulation table elements, `data` as the array storing the noise modulation table, and 0 as the number of the lookup table to set.

**Code 11-13 Loading a Noise Modulation Table**

```
glTexImage1D(GL_LUT_TEXTURE0_DMP, 0, GL_LUMINANCEF_DMP, 256, 0,
                GL_LUMINANCEF_DMP, GL_FLOAT, data);
```

The noise modulation table to use as continuous noise functions use the `glUniform1i()` function to specify the lookup table number as the following reserved uniform. Note that the specified lookup table number `GL_LUT_TEXTURE`*`i`*`_DMP`, where *`i`* represents a number from 0 to 31, does not specify the name (ID) of the texture nor `GL_LUT_TEXTURE`*`i`*`_DMP` directly.

**Table 11-25. Reserved Uniforms That Specify Noise Modulation Tables**

| Reserved Uniform | Value to Set |
|---|---|
| `dmp_Texture[3].ptSamplerNoiseMap` | Specifies the lookup table number to use as the continuous noise function.<br>0 through 31 |

To illustrate the effects of the three noise parameters F, P, and A on the output results, consider the difference between a procedural texture that is rendered as a concentric circle when it is unaffected by noise, and the same texture when its parameters are changed in both the U and V directions. These are rendered with F(x)=x as the continuous noise function.

Figure 11-9 shows the effect of changing only the A parameter. The other parameters, F and P, are set to `0.`3 and `0.0` respectively. Although the waves become more prominent as A gets larger, note that most points along the circumference of the circle are unaffected.

**Figure 11-9. Effect of the A Parameter**

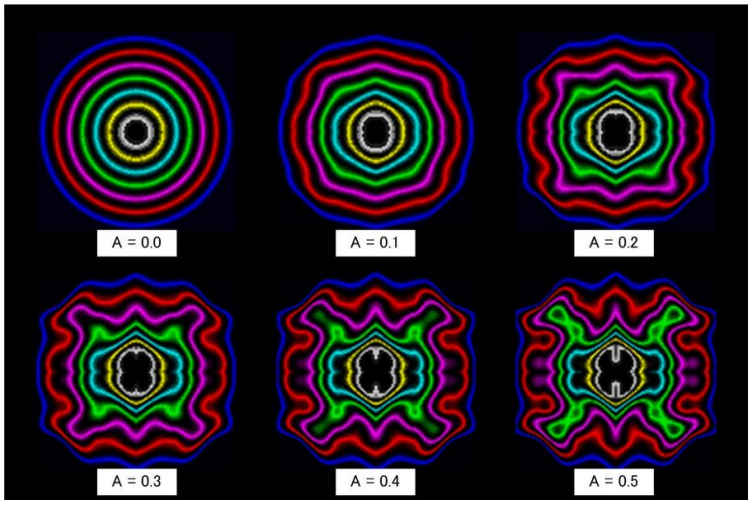



Figure 11-10 shows the effect of changing only the F parameter. The other parameters, A and P, are set to `0.`3 and `0.0` respectively. Note that as F gets larger, the frequency of the noise (fluctuations) increases, and the affected locations along the circumference of the circle get closer

to each other. Also note that the absolute value is used for the F parameter when calculating noise, so reversing the sign does not change the result.

**Figure 11-10. Effect of the F Parameter**

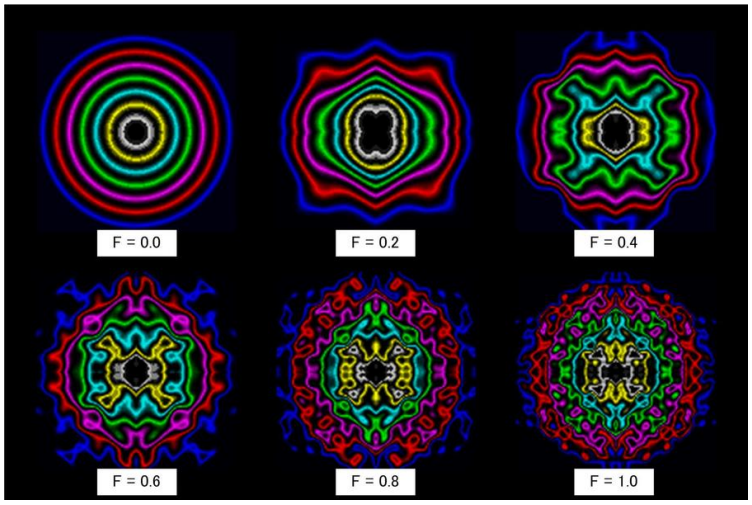
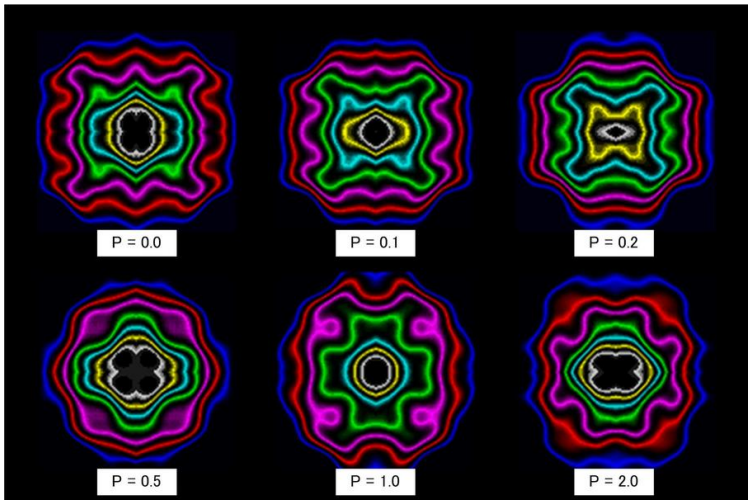



Figure 11-11 shows the effect of changing only the P parameter. The other parameters, A and F, are both set to `0.3`. Note that when P changes, only the shape of the noise changes. By modifying the P parameter, you can change a procedural texture so that it appears animated.

If you set the P parameter equal to a large value while using it to animate a texture, small changes to the P parameter will affect the shape of the noise. This is caused by the accuracy of calculations in the hardware. For example, if you animate a texture by adding a constant value to the P parameter every frame (which changes the noise), you must restore the P parameter to a small value before it gets too large. One characteristic of the F and P parameters is that when they are both positive and have a product that is a multiple of 16, they have the same effect as when the P parameter is `0.0`. In other words, by changing the P parameter back to `0` when the product of the P and F parameters is 16, you can maintain the animation's continuity. However, you may not get the same shape when the P parameter is `0.0` and when the product of the F and P parameters is 16, if the F parameter is large.

If the values for the F and A parameters are fixed, and the P parameter varies in a range where the sign does not change for the phase |u| + u (or for the phase |v| + v), you can get the same noise result for X + 16 when F×P is some arbitrary value X. However, depending on the accuracy of the noise calculation process, you may not be able to get the same random value this way if you set a large value for the F parameter. Also, if you set a large value for the P parameter, changes in small values may not be applied to the noise result.

**Figure 11-11. Effect of the P Parameter**

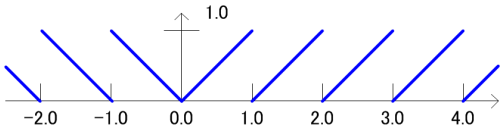
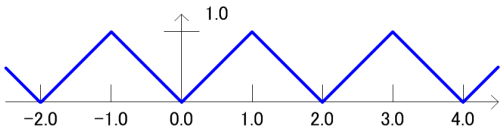
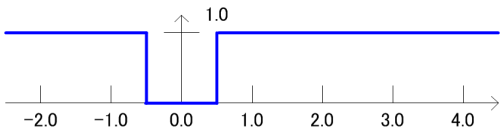
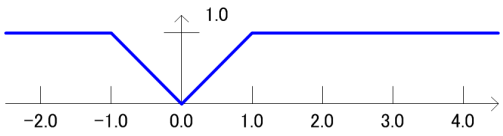
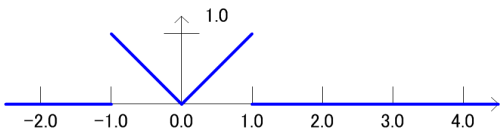



## 11.4.8. Wrapping and Symmetry Settings

Procedural textures have a feature equivalent to the wrapping mode that can be set for normal textures. This feature is called *clamp calculation*, and can be configured with dedicated modes such as *pulse* and *zero-clamp*. There are also *shift calculations* that shift blocks of texture coordinates that have the same integer values during wrapping.

Clamp calculations use the clamp mode to determine how to convert texture coordinates that are less than `0.0` or greater than `1.0` into values between `0.0` and `1.0`.

**Table 11-26. Clamp Modes**

| Clamp Mode | Coordinate Clamping |
|---|---|
| `GL_SYMMETRICAL_REPEAT_DMP` |  |
| `GL_MIRRORED_REPEAT` |  |
| `GL_PULSE_DMP` |  |
| `GL_CLAMP_TO_EDGE` (default) |  |
| `GL_CLAMP_TO_ZERO_DMP` |  |

You can set different clamp modes for the U and V texture coordinates. To set these modes, call the `glUniform1i()` function on the reserved uniform `dmp_Texture[3].ptClampU` or `dmp_Texture[3].ptClampV`.

`GL_SYMMETRICAL_REPEAT_DMP` lines up the same image on a grid. `GL_MIRRORED_REPEAT` uses a mirror reflection at even-numbered values. `GL_PULSE_DMP` uses the pixel closest to the edge of the texture for each pixel that is used for rendering. `GL_CLAMP_TO_EDGE` uses a texture's internal image for values between -1.0 and 1.0. Outside of that range, it uses pixels at the edge of the texture. `GL_CLAMP_TO_ZERO_DMP` uses the texture image for values between `–1.0` and `1.0` (excluding these two values). Outside of that range, it uses the image at coordinate 0 (including the values `–1.0` and `1.0`).

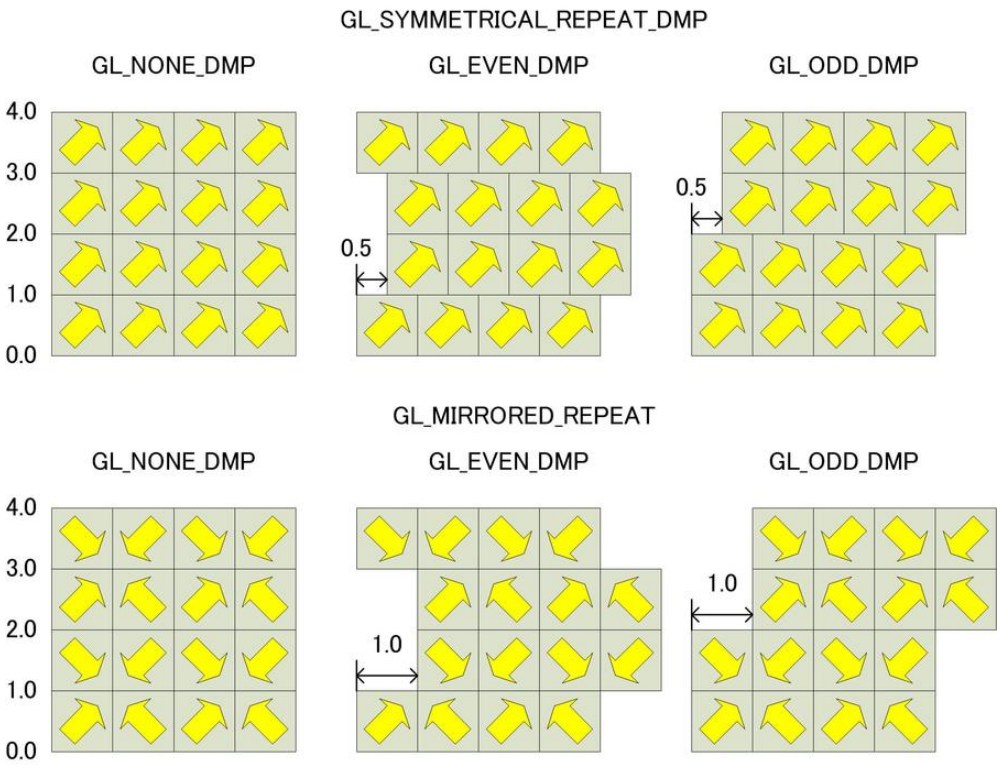
Shift calculations determine the shift coordinates based on the shift mode. The shift width depends on the clamp mode. This process is applied before clamp calculations, allowing you to avoid rendering the same image over and over.

**Table 11-27. Shift Modes**

| Shift Mode | Shift Calculation | Shift Width |
|---|---|---|
| `GL_NONE_DMP` (default) | No shift calculation. | None. |
| `GL_ODD_DMP` | Shifts coordinates when their integer value changes from an odd number to an even number. | `1.0` for `GL_MIRRORED_REPEAT` only; `0.5` otherwise. |
| `GL_EVEN_DMP` | Shifts coordinates when their integer value changes from an even number to an odd number. | `1.0` for `GL_MIRRORED_REPEAT` only; `0.5` otherwise. |

You can set different shift modes for the U and V texture coordinates. To set these modes, call the `glUniform1i()` function on the reserved uniform `dmp_Texture[3].ptShiftU` or `dmp_Texture[3].ptShiftV`.

**Figure 11-12. How Shift Modes and Clamping Modes Affect Shift Widths**



# 12. Reserved Fragment Shaders

A reserved fragment shader can process lighting and other effects for fragments output from the previous shader.

As described in 5. Shader Programs, the reserved fragment shader does not need to be loaded from a binary. To use it, attach it to the same program object as the vertex shader and geometry shader, using the special name `GL_DMP_FRAGMENT_SHADER_DMP`. The reserved fragment shader is a collection of fragment processing features. These features include the following.

- Fragment lighting
- Shadows
- Fog
- Gas
- Miscellaneous (alpha test, w buffer)

There are reserved uniforms for each fragment process. These reserved uniforms have default values and must be set, as necessary, by the application.

## 12.1. Fragment Operation Mode

Reserved fragment processing replaces the standard OpenGL fragment pipeline (from the alpha test onward) with independent processing that can handle the special rendering passes required by shadows and gas. To switch this fragment operation mode, set the reserved uniform (`dmp_FragOperation.mode`) to the desired mode using the `glUniform1i()` function.

**Table 12-1. Fragment Operation Modes**

| Fragment Operation Mode | Pipeline |
|---|---|
| `GL_FRAGOP_MODE_GL_DMP` (default) | Standard OpenGL fragment pipeline (standard mode). |
| `GL_FRAGOP_MODE_SHADOW_DMP` | Fragment pipeline for the shadow accumulation pass (shadow mode). |
| `GL_FRAGOP_MODE_GAS_ACC_DMP` | Fragment pipeline for rendering density information (gas mode). |

## 12.2. Fragment Lighting

The 3DS system uses fragment lighting, which calculates the primary and secondary colors for each fragment, rather than for each vertex. It also has the following features: bump mapping, which references a texture to perturb normal vectors; shadows, which involve the creation of shadow textures and color calculations; and attenuation, which is calculated from the distance to a spotlight

or another light.

The primary and secondary colors are determined by first combining multiple functions that output lookup table values based on the dot product of two vectors, and then using bitwise AND/OR operations to combine those output values. With 3DS, although you cannot fully customize the method that combines the lookup tables and vectors for the dot products, you can select different configurations from preset settings.

Nintendo expects the CTR system to use eye coordinates for several vectors, because lighting equations are considered to be in eye coordinates. Although eye coordinates are not necessarily required, all the vectors that are used must be in the same coordinate system.

To use fragment lighting, you must set `dmp_FragmentLighting.enabled` to `GL_TRUE` with the `glUniform1i()` function and enable at least one light. In the vertex shader, the normal vector, view vector, and tangent vector (when required for lighting) must be converted into a quaternion and output as a single vertex attribute.

**Table 12-2. Enabling and Disabling Fragment Lighting**

| Reserved Uniform | Type | Setting Value |
|---|---|---|
| `dmp_FragmentLighting.enabled` | `bool` | `GL_TRUE`: Enable lighting.<br>`GL_FALSE`: Disable lighting (default). |

## 12.2.1. Quaternion Conversion

Lighting calculations require all vectors to use the same coordinate system. In other words, bump mapping (described later) must convert the perturbation normals referenced in a texture from surface-local coordinates (with the vertex at the origin and the normal vector along the positive z-axis) into eye coordinates.

The following 3×3 rotation matrix converts surface-local coordinates into eye coordinates. It comprises a normal, tangent, and binormal vector, and can be converted into a quaternion (Qx, Qy, Qz, Qw).

$$\begin{bmatrix} Ex \\ Ey \\ Ez \end{bmatrix} = \begin{bmatrix} Tx & Bx & Nx \\ Ty & By & Ny \\ Tz & Bz & Nz \end{bmatrix} \begin{bmatrix} Sx \\ Sy \\ Sz \end{bmatrix}$$

(E represents the eye coordinates, T is the tangent, N is the normal, B is the binormal, and S represents the surface-local coordinates.)

$$Qw = \frac{1}{2}\sqrt{1 + Tx + By + Nz}$$

$$Qx = \frac{1}{4Qw}(Bz - Ny)$$

$$Qy = \frac{1}{4Qw}(Nx - Tz)$$

$$Qz = \frac{1}{4Qw}(Ty - Bx)$$

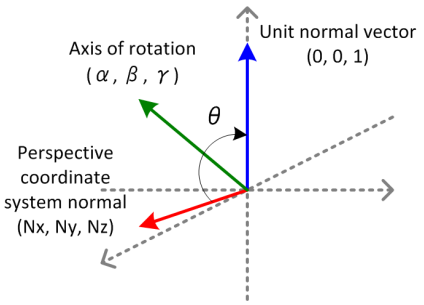
Fragment lighting is implemented to generate a quaternion for each fragment from a quaternion for each vertex, rather than generate a vector for each fragment from the normal, tangent, and binormal (which can be calculated from the normal and tangent) vectors input for each vertex. The quaternion is converted into the original rotation matrix during fragment light processing. To use

fragment lighting, you must convert each vector into a valid quaternion in the vertex shader.

### Generating Normal-Only Quaternions

The CTR-SDK sample demos include a vertex shader (`l_position_view_quaternion`) that generates quaternions using only the vertex normal information. To generate a quaternion that transforms normals in surface-local coordinates to normals in perspective coordinates, the vertex shader uses the half angle vector between the unit normal vector (0, 0, 1) and a normal vector that has been transformed to perspective coordinates (Nx, Ny, Nz) as the axis of rotation, and derives a quaternion that performs 180° rotations.

**Figure 12-1. The Rotation Axis and Angle of a Quaternion**




For an axis of rotation (α, β, γ) and an angle of rotation θ, the derived quaternion Q would be computed as follows.

$$Q = (\cos\frac{\theta}{2}; \alpha\sin\frac{\theta}{2}, \beta\sin\frac{\theta}{2}, \gamma\sin\frac{\theta}{2})$$

**Note:** In the vertex shader, the real component of the quaternion is set to the w component.

For a θ value of 180°, this becomes:

Q = ( 0; α, β, γ )

Taking the half-angle vector as the axis of rotation, this becomes:

$$\begin{aligned}
&(\alpha,\beta,\gamma)\\
&=\left(\frac{Nx+0}{\sqrt{(Nx+0)^2+(Ny+0)^2+(Nz+1)^2}}, \frac{Ny+0}{\sqrt{(Nx+0)^2+(Ny+0)^2+(Nz+1)^2}}, \frac{Nz+1}{\sqrt{(Nx+0)^2+(Ny+0)^2+(Nz+1)^2}}\right)\\
&=\left(\frac{Nx}{\sqrt{Nx^2+Ny^2+Nz^2+2Nz+1}}, \frac{Ny}{\sqrt{Nx^2+Ny^2+Nz^2+2Nz+1}}, \frac{Nz+1}{\sqrt{Nx^2+Ny^2+Nz^2+2Nz+1}}\right)\\
&=\left(\frac{Nx}{\sqrt{2(Nz+1)}}, \frac{Ny}{\sqrt{2(Nz+1)}}, \frac{\sqrt{Nz+1}}{\sqrt{2}}\right)
\end{aligned}$$

The orientation of the half-angle vector is undefined only when (Nx, Ny, Nz) is (0, 0, –1). In this case, assume that (α, β, γ) = (1, 0, 0).

## 12.2.2. Lighting Overview

3DS fragment lighting always calculates the primary and secondary colors.

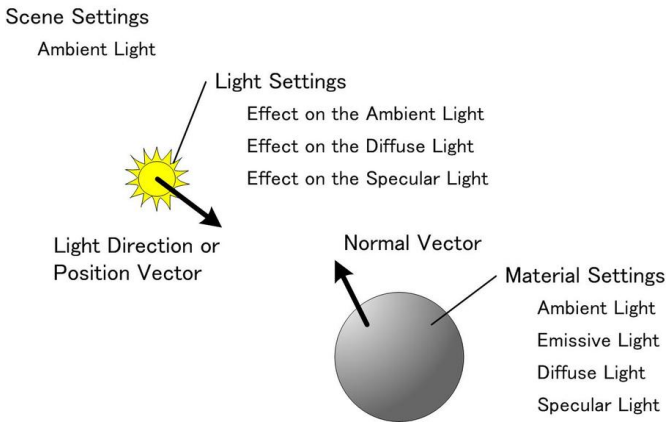
The primary color is calculated first by accumulating each light's effect (with shadows, spotlight

attenuation, and distance attenuation applied) on a fragment's ambient and diffuse light. The fragment's emissive light and the effect of the scene's ambient light is then added to this accumulated value. This becomes the fragment's base color.

The secondary color is calculated by accumulating each light's effect on a fragment's second specular light with shadows, spotlight attenuation, and distance attenuation applied. This color is mainly used for fragment highlights.

Like OpenGL, an object's color is determined from its ambient, diffuse, emissive, and specular light. Unlike OpenGL, however, lighting uses per-fragment calculations and second specular light, making it possible to calculate the specular light in a variety of ways. The second specular light in particular can be used to represent materials with colors that change depending on the angle.

**Figure 12-2. Fragment Lighting**




## 12.2.3. Scene Settings

Fragment lighting can handle scene sizes from $-2^{16}$ through $2^{15}$. Do not allow the distance between the viewpoint and any fragment or light in the scene to be greater than or equal to $2^{16}$.

The scene affects fragments through its ambient light (the global ambient light). To specify the scene's global ambient light, set the reserved uniform `dmp_FragmentLighting.ambient` to an RGBA color using the `glUniform4fv()` function.

**Table 12-3. Reserved Uniforms for Scene Settings**

| Reserved Uniform | Type | Setting Value |
|---|---|---|
| `dmp_FragmentLighting.ambient` | `vec4` | Specifies the scene's global ambient light (R, G, B, A). Each component has a value between 0.0 and 1.0. This is (0.2, 0.2, 0.2, 1.0) by default. |

## 12.2.4. Material Settings

Material settings can be described simply as settings that use color information, such as the ambient and specular light, to represent a fragment's materials and texture. Specify material-related settings in the reserved uniforms `dmp_FragmentMaterial.*`.

12.2.7. Equations for the Primary Color and 12.2.8. Equations for the Secondary Color describe how the settings are used in lighting calculations.

**Table 12-4. Reserved Uniforms for Material Settings**

| Reserved Uniform | Type | Setting Value |
|---|---|---|
| `dmp_FragmentMaterial.ambient` | `vec4` | Specifies the ambient light (R, G, B, A). Each component has a value between 0.0 and 1.0.<br>This is (0.2, 0.2, 0.2, 1.0) by default. |
| `dmp_FragmentMaterial.diffuse` | `vec4` | Specifies the ambient light (R, G, B, A). Each component has a value between 0.0 and 1.0.<br>This is (0.8, 0.8, 0.8, 1.0) by default. |
| `dmp_FragmentMaterial.emission` | `vec4` | Specifies the ambient light (R, G, B, A). Each component has a value between 0.0 and 1.0.<br>This is (0.0, 0.0, 0.0, 1.0) by default. |
| `dmp_FragmentMaterial.specular0` | `vec4` | Specifies specular light 0 (R, G, B, A). Each component has a value between 0.0 and 1.0.<br>This is (0.0, 0.0, 0.0, 1.0) by default. |
| `dmp_FragmentMaterial.specular1` | `vec4` | Specifies specular light 1 (R, G, B, A). Each component has a value between 0.0 and 1.0.<br>This is (0.0, 0.0, 0.0, 1.0) by default. |
| `dmp_FragmentMaterial.samplerXX` (XX=D0,D1,RR,RG,RB,FR) | `int` | Specifies the lookup table numbers to use for lighting calculations.<br>Each factor is a number between 0 and 31. |

## 12.2.5. Light Settings

There are two types of light settings. One configures the effect of light on a material and the other configures the light itself. Fragment lighting can handle eight lights. Specify light-related settings in the reserved uniforms (`dmp_FragmentLightSource[`*i*`].`∗, where *i* is the light number between 0 and 7).

12.2.7. Equations for the Primary Color and 12.2.8. Equations for the Secondary Color describe how the settings are used in lighting calculations.

**Table 12-5. Reserved Uniforms for Light Settings**

| Reserved Uniform | Type | Setting Value |
|---|---|---|
| ∗`.enabled` | `bool` | Enables or disables a light.<br>`GL_TRUE`: Enable light.<br>`GL_FALSE`: Disable light (default). |
| ∗`.ambient` | `vec4` | Specifies an ambient light (R, G, B, A). Each component has a value between 0.0 and 1.0.<br>This is (0.0, 0.0, 0.0, 0.0) by default. |
| ∗`.diffuse` | `vec4` | Specifies a diffuse light (R, G, B, A). Each component has a value between 0.0 and 1.0.<br>By default, only light number 0 is (1.0, 1.0, 1.0, 1.0).<br>All of the others are (0.0, 0.0, 0.0, 0.0). |
| ∗`.specular0` | `vec4` | Specifies specular light 0 (R, G, B, A). Each component has a value between 0.0 and 1.0.<br>By default, only light number 0 is (1.0, 1.0, 1.0, 1.0).<br>All of the others are (0.0, 0.0, 0.0, 0.0). |
| ∗`.specular1` | `vec4` | Specifies specular light 1 (R, G, B, A). Each component has a value between 0.0 and 1.0.<br>This is (0.0, 0.0, 0.0, 0.0) by default. |
| ∗`.position` | `vec4` | Specifies the position of a light source (x, y, z, w).<br>The vector does not need to be normalized. The w component is used to distinguish between directional (0.0) and positional light sources.<br>This is (0.0, 0.0, 1.0, 0.0) by default. |

| | | |
|---|---|---|
| ∗.spotDirection | vec3 | Specifies a spotlight direction (x, y, z).<br>The vector does not need to be normalized.<br>(0.0, 0.0, –1.0) by default. |
| ∗.shadowed | bool | Specifies whether a light is affected by shadows.<br>GL_TRUE: Affected by shadows.<br>GL_FALSE: Not affected by shadows (default). |
| ∗.geomFactor0 | bool | Specifies whether to use geometry factor 0 in lighting calculations.<br>GL_TRUE: Use geometry factor 0.<br>GL_FALSE: Do not use geometry factor 0 (default). |
| ∗.geomFactor1 | bool | Specifies whether to use geometry factor 1 in lighting calculations.<br>GL_TRUE: Use geometry factor 1.<br>GL_FALSE: Do not use geometry factor 1 (default). |
| ∗.twoSideDiffuse | bool | Specifies whether to use two-sided lighting.<br>GL_TRUE: Use two-sided lighting.<br>GL_FALSE: Do not use two-sided lighting (default). |
| ∗.spotEnabled | bool | Specifies whether to attenuate a spotlight based on its light distribution.<br>GL_TRUE: Apply spotlight attenuation.<br>GL_FALSE: Do not apply spotlight attenuation (default). |
| ∗.distanceAttenuationEnabled | bool | Specifies whether to attenuate a light over distance.<br>GL_TRUE: Apply distance attenuation.<br>GL_FALSE: Do not apply distance attenuation. |
| ∗.distanceAttenuationBias | float | Specifies the distance attenuation bias.<br>0.0 by default. |
| ∗.distanceAttenuationScale | float | Specifies the distance attenuation scale.<br>1.0 by default. |
| ∗.samplerXX<br>(XX=SP,DA) | int | Specifies the lookup table number to use for calculating spotlight attenuation and distance attenuation.<br>Each factor is a number between 0 and 31. |

The table uses asterisks (∗) to indicate dmp_FragmentLightSource[*i*]. Where *i* is a light number between 0 and 7.

## 12.2.6. Lighting Environment

The reserved uniforms dmp_LightEnv.∗ configure settings related to general lighting, including shadow texture selection, bump map settings, and lookup table input.

12.2.7. Equations for the Primary Color and 12.2.8. Equations for the Secondary Color describe how the settings are used in lighting calculations.

**Table 12-6. Reserved Uniforms for the Lighting Environment**

| Reserved Uniform | Type | Setting Value |
|---|---|---|
| ∗.absLutInputXX<br>(XX=D0,D1,RR,RG,RB,FR,SP) | bool | Specifies whether to convert lookup table input for each factor into absolute values.<br>GL_TRUE: Convert to absolute values.<br>GL_FALSE: Do not convert to absolute values (default). |
| | | Specifies the cosine of the angle between two vectors to use as lookup table input for each factor.<br>GL_LIGHT_ENV_NH_DMP: The normal and half vectors (default). |

| | | |
|---|---|---|
| *.lutInputXX (XX=D0,D1,RR,RG,RB,FR,SP) | int | GL_LIGHT_ENV_VH_DMP: The view and half vectors.<br>GL_LIGHT_ENV_NV_DMP: The normal and view vectors.<br>GL_LIGHT_ENV_LN_DMP: The light and normal vectors.<br>GL_LIGHT_ENV_SP_DMP: The light and spotlight vectors.<br>GL_LIGHT_ENV_CP_DMP: The tangent vector and the projection of the half vector onto the tangent plane. |
| *.lutScaleXX (XX=D0,D1,RR,RG,RB,FR,SP) | float | Specifies the scale to apply to each factor's lookup table output. After applying a scale value to the value output by the lookup table, clamping is performed within a range from -2.0 through 2.0.<br>0.25<br>0.5<br>1.0 (default)<br>2.0<br>4.0<br>8.0 |
| *.shadowSelector | int | Specifies the texture unit to use for shadows.<br>GL_TEXTURE0 (default)<br>GL_TEXTURE1<br>GL_TEXTURE2<br>GL_TEXTURE3 |
| *.bumpSelector | int | Specifies the texture unit to use for bump maps.<br>GL_TEXTURE0 (default)<br>GL_TEXTURE1<br>GL_TEXTURE2<br>GL_TEXTURE3 |
| *.bumpMode | int | Specifies the perturbation mode for normal and tangent vectors.<br>GL_LIGHT_ENV_BUMP_NOT_USED_DMP: No bump mapping (default).<br>GL_LIGHT_ENV_BUMP_AS_BUMP_DMP: Perturb normal vectors.<br>GL_LIGHT_ENV_BUMP_AS_TANG_DMP: Perturb tangent vectors. |
| *.bumpRenorm | bool | Specifies whether to regenerate the third component of the normal vector.<br>GL_TRUE: Regenerate.<br>GL_FALSE: Do not regenerate (default). |
| *.config | int | Specifies the configuration for each factor.<br>GL_LIGHT_ENV_LAYER_CONFIG0_DMP (default)<br>GL_LIGHT_ENV_LAYER_CONFIG1_DMP<br>GL_LIGHT_ENV_LAYER_CONFIG2_DMP<br>GL_LIGHT_ENV_LAYER_CONFIG3_DMP<br>GL_LIGHT_ENV_LAYER_CONFIG4_DMP<br>GL_LIGHT_ENV_LAYER_CONFIG5_DMP<br>GL_LIGHT_ENV_LAYER_CONFIG6_DMP<br>GL_LIGHT_ENV_LAYER_CONFIG7_DMP |
| *.invertShadow | bool | Specifies whether to invert the shadow term (1.0 - shadow).<br>GL_TRUE: Invert.<br>GL_FALSE: Do not invert (default). |
| *.shadowPrimary | bool | Specifies whether to apply shadows to the primary color.<br>GL_TRUE: Apply.<br>GL_FALSE: Do not apply (default). |
| *.shadowSecondary | bool | Specifies whether to apply shadows to the secondary color.<br>GL_TRUE: Apply.<br>GL_FALSE: Do not apply (default). |
| *.shadowAlpha | bool | Specifies whether to apply shadows to the alpha component.<br>GL_TRUE: Apply.<br>GL_FALSE: Do not apply (default). |
| | | Specifies the output mode for the Fresnel factor.<br>GL_LIGHT_ENV_NO_FRESNEL_DMP (default) |

| | | |
|---|---|---|
| *.fresnelSelector | int | GL_LIGHT_ENV_PRI_ALPHA_FRESNEL_DMP<br>GL_LIGHT_ENV_SEC_ALPHA_FRESNEL_DMP<br>GL_LIGHT_ENV_PRI_SEC_ALPHA_FRESNEL_DMP |
| *.clampHighlights | bool | Specifies whether to clamp the specular output value.<br>GL_TRUE: Clamp.<br>GL_FALSE: Do not clamp (default). |
| *.lutEnabledD0 | bool | Specifies whether to apply output values from the lookup table for distribution 0 (D0).<br>GL_TRUE: Apply.<br>GL_FALSE: Do not apply (default). |
| *.lutEnabledD1 | bool | Specifies whether to apply output values from the lookup table for distribution 1 (D1).<br>GL_TRUE: Apply.<br>GL_FALSE: Do not apply (default). |
| *.lutEnabledRefl | bool | Specifies whether to apply output values from the lookup tables for reflection (RR, RG, RB).<br>GL_TRUE: Apply.<br>GL_FALSE: Do not apply (default). |

The table uses asterisks (*) to indicate dmp_LightEnv.

## 12.2.7. Equations for the Primary Color

The following formula summarizes how the primary color is calculated.

$$Color_{primary} = \sum((Diffuse \times DP_{LN} \times Shadow + Ambient) \times Spot \times DistAtt + Ambient_{global} + Emission$$

The following paragraphs provide more information about how each term is calculated.

The product of the material and light color components is applied to the diffuse and ambient light.

$$Diffuse = Diffuse_{material} \times Diffuse_{light}$$
$$Ambient = Ambient_{material} \times Ambient_{light}$$

The diffuse light is affected by shadows and the angle of incident light.

$$DP_{LN} = \max \{ 0, L \cdot N \} \text{ or } \text{abs} ( L \cdot N )$$

The effect of the angle of incident light is specified by $DP_{LN}$ in the equation. It is the dot product of a normalized light vector and normal vector. For one-sided lighting, it is the larger of 0 and the dot product, and for two-sided lighting it is the absolute value of the dot product. To enable one-sided or two-sided lighting, set dmp_FragmentLightSource[*i*].twoSideDiffuse to GL_FALSE or GL_TRUE respectively.

The effect of shadow attenuation is represented by *Shadow* in the equation. If either dmp_LightEnv.shadowPrimary or dmp_FragmentLightSource[*i*].shadowed is GL_FALSE, shadows have no effect and a value of 1.0 is applied. If either reserved uniform is GL_TRUE, the value used is sampled from the texture unit specified by dmp_LightEnv.shadowSelector. If dmp_LightEnv.invertShadow is GL_TRUE, the value used is actually the sampled value subtracted from 1.0. If any texture other than a shadow texture is bound to the specified texture unit, color components are applied unchanged to the sampled value.

The effect of spotlights is represented by *Spot* in the equation. You can set a lookup table for each light and configure whether it is used. Use dmp_FragmentLightSource[*i*].spotEnabled to configure whether spotlights are used and dmp_FragmentLightSource[*i*].samplerSP to set the lookup tables used by spotlights. Use dmp_FragmentLightSource[*i*].spotDirection to set the spotlight direction vector. A value of GL_LIGHT_ENV_SP_DMP is usually specified as the lookup table input for spotlights (dmp_LightEnv.lutInputSP).

The effect of distance attenuation is represented by *DistAtt* in the equation. Directional light sources are unaffected by distance attenuation.

The reserved uniform `dmp_FragmentLightSource[i].distanceAttenuationEnabled` configures whether distance attenuation is applied. This can be controlled for each light. However, the effect of distance attenuation is disabled (`1.0` is applied) when `dmp_LightEnv.config` is `GL_LIGHT_ENV_LAYER_CONFIG7_DMP`.

The lookup table to use is set by `dmp_FragmentLightSource[i].samplerDA`. Lookup table input is affected by the values set for the scale (`dmp_FragmentLightSource[i].distanecAttenuationScale`) and bias (`dmp_FragmentLightSource[i].distanceAttenuationBias`).

This section has so far described the effect of each light on a fragment's primary color. This effect is calculated only for valid lights and then it is added to the material's emissive color and the global ambient color, which are unaffected by lights, to compute the fragment's final primary color.

The global ambient light is the product of the material's ambient light and the scene's ambient light.

$$Ambient_{global} = Ambient_{material} \times Ambient_{scene}$$

When light source 0 is disabled (`dmp_FragmentLightSource[0].enabled` is `GL_FALSE`), however, a value of `0.0` is applied to the global ambient light and the material's emissive light.

## 12.2.8. Equations for the Secondary Color

The following formula summarizes how the secondary color is calculated.

$$Color_{secondary} = \sum((Specular0 + Specular1) \times f \times Shadow \times Spot \times DistAtt)$$

The following paragraphs provide more information about how each term is calculated.

Except that `dmp_LightEnv.shadowSecondary` settings are used instead of `dmp_LightEnv.shadowPrimary` settings, *Shadow*, *Spot*, and *DistAtt* are each calculated in the same way that they are calculated for the primary color.

*Specular0* and *Specular1* are each calculated as follows.

$$Specular0 = Specular0_{material} \times Specular0_{light} \times Distribution0 \times Geometry0$$
$$Specular1 = Reflection_{RGB} \times Specular1_{light} \times Distribution1 \times Geometry1$$

The specular term is usually calculated as the product of the specular property of a material, the specular color of a light, a distribution function, and a geometry factor. Some settings allow output from lookup tables that define different reflections for each color to be applied to the term that corresponds to a material's specular light 1. By adjusting the reflection and distribution lookup tables, you can represent fragments with a variety of different textures.

Distribution functions (factors) are represented by *Distribution0* and *Distribution1* in the equation. The distribution functions are configured through lookup tables for distribution 0 (D0) and distribution 1 (D1).

The reserved uniform `dmp_LightEnv.lutEnabledD0` (`lutEnabledD1`) controls whether these functions are used. The reserved uniform `dmp_FragmentLightSource[i].samplerD0` (`samplerD1`) specifies the lookup table number to use. The reserved uniforms `dmp_LightEnv.lutInputD0` and `dmp_LightEnv.lutInputD1` specify the lookup table input. `dmp_LightEnv.absLutInputD0` and `dmp_LightEnv.absLutInputD1` specify the absolute value of the input. Lookup table output accounts for the scale values specified by `dmp_LightEnv.lutScaleD0` and `dmp_LightEnv.lutScaleD1`.

Reflections are represented by $Reflection_{RGB}$ in the equation. They use lookup tables (RR, RG, RB) to set functions that calculate the reflection for each RGB component instead of a material's specular light 1. The reserved uniform `dmp_LightEnv.lutEnabledRefl` controls whether this

feature is used. If GL_FALSE is specified, the material's specular color 1 is applied. The reserved uniform dmp_FragmentLightSource[*i*].samplerRR (samplerRG, samplerRB) specifies the lookup table number to use. The reserved uniforms dmp_LightEnv.lutInputXX specify the lookup table input and dmp_LightEnv.absLutInputXX specify the absolute value of the input (where XX is RR, RG, or RB).

Geometry factors are represented by *Geometry0* and *Geometry1* in the equation. They are used by the Cook-Torrance lighting model. The reserved uniforms dmp_FragmentLightSource[*i*].geomFactor0 and dmp_FragmentLightSource[*i*].geomFactor1 control whether they are used. A value of 1.0 is applied for a setting of GL_FALSE and an approximation of the geometry factors used in the Cook-Torrance lighting model is applied for a setting of GL_TRUE.

*f* is a function that uses the dot product of the normalized light vector and normal vector to determine whether lighting is enabled for a fragment. A value of 1.0 is always applied when dmp_LightEnv.clampHighlights is set to GL_FALSE. This setting is used to represent translucent objects that allow light to pass through them to areas where it would not otherwise reach. If GL_TRUE is specified, a value of 0.0 is applied when the dot product is 0.0 or less and a value of 1.0 is applied when the dot product is greater than 0.0. In this case, unlit areas have no specular light.

A fragment's final secondary color is calculated from the effect of each valid light's effect on it.

## 12.2.9. Alpha Component Lighting

The previous sections described how to calculate the primary and secondary colors with the alpha component fixed at 1.0. Fragment lighting allows you to apply Fresnel factors and shadows to the alpha component.

Fresnel factors were originally intended to be used as lookup tables (FR) for Fresnel reflections in translucent objects, but by replacing the alpha component with lookup table output, they can also be used for other purposes.

The reserved uniform dmp_FragmentLightSource[*i*].samplerFR specifies the number of the lookup table to use for the Fresnel factors. The reserved uniform dmp_LightEnv.lutInputFR specifies the lookup table input and dmp_LightEnv.absLutInputFR specifies the absolute values of the input. If multiple lights have been enabled, the light with the largest number has its light vector used in the dot product that is input to the lookup table. The reserved uniform dmp_LightEnv.fresnelSelector uses the following values to control the extent to which Fresnel factors are applied.

**Table 12-7. Scope of the Applied Fresnel Factor**

| Setting Value | Applies To |
|---|---|
| GL_LIGHT_ENV_NO_FRESNEL_DMP (default) | Nothing. (Fix the alpha component at 1.0.) |
| GL_LIGHT_ENV_PRI_ALPHA_FRESNEL_DMP | Only the alpha component for the primary color. |
| GL_LIGHT_ENV_SEC_ALPHA_FRESNEL_DMP | Only the alpha component for the secondary color. |
| GL_LIGHT_ENV_PRI_SEC_ALPHA_FRESNEL_DMP | The alpha component for the primary and secondary colors. |

The Fresnel factor is also applied to the alpha component for shadows. The shadow alpha component is multiplied with the applied alpha component when a value of GL_TRUE is specified for dmp_LightEnv.shadowAlpha, which controls the effect on the shadow alpha component. If dmp_LightEnv.invertShadow is GL_TRUE, the shadow alpha value is subtracted from 1.0 (as it is for colors) before being multiplied.

## 12.2.10. Creating and Specifying Lookup Tables

Lighting equations use the following eight types of lookup tables.

- Reflections (three types: RR, RG, and RB)
- Distribution factors (two types: D0 and D1)
- Fresnel factors (FR)
- Spotlights (SP)
- Distance attenuation of light

The lookup tables for reflections (RR, RG, RB), distribution factors (D0, D1), and Fresnel factors are all material settings, and are common to all lights. The lookup tables for spotlights (SP) and the distance attenuation of light (DA) can be set differently for each light.

The layer configuration (`dmp_LightEnv.config`) can control which lookup tables are used for each term in the secondary color's lighting equation, except for the distance attenuation of light.

**Table 12-8. Lookup Tables and Number of Cycles for Each Layer Configuration**

| Layer Configuration | Rr | Rg | Rb | D0 | D1 | Fr | Sp | Cycles |
|---|---|---|---|---|---|---|---|---|
| GL_LIGHT_ENV_LAYER_CONFIG0_DMP | RR | RR | RR | D0 | - | - | SP | 1 |
| GL_LIGHT_ENV_LAYER_CONFIG1_DMP | RR | RR | RR | - | - | FR | SP | 1 |
| GL_LIGHT_ENV_LAYER_CONFIG2_DMP | RR | RR | RR | D0 | D1 | - | - | 1 |
| GL_LIGHT_ENV_LAYER_CONFIG3_DMP | - | - | - | D0 | D1 | FR | - | 1 |
| GL_LIGHT_ENV_LAYER_CONFIG4_DMP | RR | RG | RB | D0 | D1 | - | SP | 2 |
| GL_LIGHT_ENV_LAYER_CONFIG5_DMP | RR | RG | RB | D0 | - | FR | SP | 2 |
| GL_LIGHT_ENV_LAYER_CONFIG6_DMP | RR | RR | RR | D0 | D1 | FR | SP | 2 |
| GL_LIGHT_ENV_LAYER_CONFIG7_DMP | RR | RG | RB | D0 | D1 | FR | SP | 4 |

The table shows which lookup tables are used to get values for a reflection's RGB components, the distribution factors, the Fresnel factors, and the spotlight term. A value of `1.0` is applied to the lighting equation for any cell that contains a hyphen (-). In other words, the corresponding term disappears from the equation. The **Cycles** column shows the number of hardware cycles required for lighting calculations. To speed up lighting calculations, choose a layer configuration that minimizes this number.

> **Warning:** When only write access to the color buffer has been configured (when the `glColorMask()` function has set a value of `GL_TRUE` for all color buffer components and the `glDisable()` function has disabled `GL_BLEND` and `GL_COLOR_LOGIC_OP`), layer configurations from `GL_LIGHT_ENV_LAYER_CONFIG4_DMP` through `GL_LIGHT_ENV_LAYER_CONFIG6_DMP` require three rather than two cycles to process a single pixel.
>
> A setting of `GL_LIGHT_ENV_LAYER_CONFIG7_DMP` disables the effect of distance attenuation when the primary and secondary color are calculated.

For example, when `GL_LIGHT_ENV_LAYER_CONFIG0_DMP` is set, all of the reflection RGB components are obtained from the RR lookup table, the distribution 0 values are obtained from the D0 lookup table, and the spotlight values are obtained from the SP lookup table. A fixed value of `1.0` is applied for distribution 1 and the Fresnel factor.

As described in 7.7. Loading Lookup Tables, lookup tables are prepared by the `glTexImage1D()` function. The lookup tables used for fragment lighting have a fixed *`width`* of 512 elements. The first 256 elements store the lookup table's sampling values and the last 256 elements store the differences between each of the sampling values.

The order that sampling values are stored in depends on whether lookup table input is between `0.0` and `1.0` or between `–1.0` and `1.0`. The reserved uniforms `dmp_LightEnv.absLutInputXX` (where XX is `D0`, `D1`, `RR`, `RG`, `RB`, `FR`, or `SP`) set the range of input values from `0.0` to `1.0` when `GL_TRUE` is specified or from `–1.0` to `1.0` when `GL_FALSE` is specified.

Procedures for getting sampling values are described next, followed by the corresponding procedures for storing them.

If the range of input values is between `0.0` and `1.0`, each input value is multiplied by 256 and then clamped to 255. The integer portion of this number is the index for getting values. The index is first used to get a sampling value from the lookup table, and then it is incremented by 256 to get a difference value. The difference value is multiplied by the fractional portion of the input value, and then added to the original sampling value to find the final sampling value.

**Code 12-1. Procedure to Get Sampling Values for Input Between 0.0 and 1.0 (Pseudocode)**

```
index=min(floor(input * 256), 255);
samplingValue=LUT[index] + LUT[index + 256] * (input * 256 - index);
```

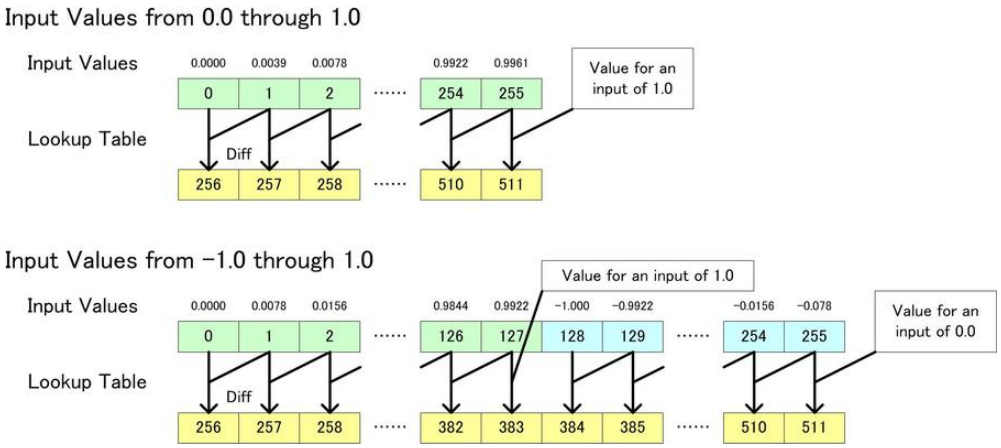
If the range of input values is between `–1.0` and `1.0`, each input value is multiplied by 128 and then its integer portion is converted to a two's complement index. The index is first used to get a sampling value from the lookup table, and then it is incremented by 256 to get a difference value. The difference value is multiplied by the fractional portion of the input value, and then added to the original sampling value to find the final sampling value.

**Code 12-2. Procedure to Get Sampling Values for Input Between –1.0 and 1.0 (Pseudocode)**

```
if (input < 0.0) {
    flooredInput=floor(input * 128);
    index = 255 + flooredInput;
    samplingValue=LUT[index] + LUT[index + 256] * (input * 128 - flooredInput);
} else {
    index=min(floor(input * 128), 127);
    samplingValue=LUT[index] + LUT[index + 256] * (input * 128 - index);
}
```

Figure 12-3 shows the order that sampling values are stored in.

**Figure 12-3. Order of Sampling Values Stored in the Lookup Table**




Lookup tables are usually created by storing the sampling value that results from dividing the index

by 256 or 128. The difference between each sampling value and the next is stored 256 indices later. Note that the lookup table is discontinuous for input values between –1.0 and 1.0.

The following pseudocode samples illustrate this process using `func` as the function that calculates sampling values from input values.

**Code 12-3. Creating a Lookup Table for Input Between 0.0 and 1.0 (Pseudocode)**

```
for (i = 0; i < 256; i++) LUT[i] = func((float) i / 256.0f);
for (i = 0; i < 255; i++) LUT[i + 256] = LUT[i + 1] - LUT[i];
LUT[511] = func(1.0f) - LUT[255] * 16.0f / 15.0f;
```

The difference value for an input of 1.0 (the last difference element) is multiplied by 16.0/15.0 because the GPU has a fractional precision of 4 bits. If the original difference value were stored, the highest input value would not produce the sampling value for 1.0.

**Code 12-4. Creating a Lookup Table for Input Between –1.0 and 1.0 (Pseudocode)**

```
for (i = 0; i < 128; i++) {
    LUT[i] = func((float) i / 128.0f);
    LUT[255 - i] = func((float) (i + 1) * -1.0f / 128.0f);
}
for (i= 0; i < 127; i++) {
    LUT[i + 256] = LUT[i + 1] - LUT[i];
    LUT[i + 384] = LUT[i + 129] - LUT[i + 128];
}
LUT[383] = func(1.0f) - LUT[127] * 16.0f / 15.0f;
LUT[511] = LUT[0] - LUT[255];
```

In this example, the value is multiplied by 16.0/15.0 because, if the original difference value were stored, the highest input value would not produce the sampling value for 1.0.

Load the created lookup table using the `glTexImage1D()` function. For *`target`* for the `glBindTexture()` function, specify *GL_LUT_TEXTUREi_DMP* to specify the lookup table to load. To reference the lookup table during lighting calculations, use the `glUniform1i()` function to specify the lookup table number as the reserved uniform. Note that the lookup table number specified here, `GL_LUT_TEXTURE`*`i`*`_DMP`, where *i* represents a number from 0 to 31, does not specify the name (ID) of the texture nor `GL_LUT_TEXTURE`*`i`*`_DMP` directly. The reserved uniform specified by the lookup table number will be either material settings when referring to reflections, distribution factors, or Fresnel factors (`dmp_FragmentMaterial.samplerXX` (where XX=D0,D1,RR,RG,RB,FR)) or light settings when referring to spotlight and light distance attenuation (`dmp_FragmentLightSource[`*`i`*`].samplerXX` (where XX=SP,DA)).

**Code 12-5. Specifying the Reflection Lookup Table (for the R Component)**

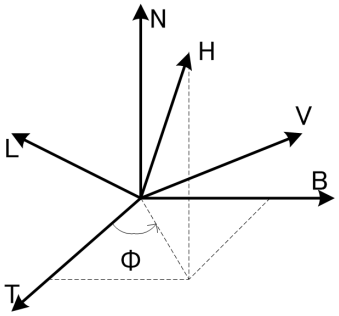
```
glBindTexture(GL_LUT_TEXTURE2_DMP, lutTextureID);
glTexImage1D(GL_LUT_TEXTURE2_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
             GL_LUMINANCEF_DMP, GL_FLOAT, LUT);
glUniform1i(glGetUniformLocation(progID, "dmp_FragmentMaterial.samplerRR"), 2);
```

## 12.2.11. Lookup Table Input

All terms (D0, D1, RR, RG, RB, FR, SP), except for the distance attenuation of light (DA), take as input the cosine of the angle between two vectors (the dot product of two normalized vectors).

The input vectors are the normal vector (N), light vector (L), view vector (V), half vector (H), tangent vector (T), binormal vector (B), spotlight direction vector, and the projection of the half vector onto the tangent plane.

**Figure 12-4. Vectors Used as Lookup Table Input**




To specify the values used as lookup table input for each factor, set the reserved uniforms for the lighting environment, `dmp_LightEnv.lutInputXX` (where XX is `D0`, `D1`, `RR`, `RG`, `RB`, `FR`, or `SP`), to one of the following values using the `glUniform1i()` function. The cosine of the angle between the two specified vectors is used as lookup table input for each factor.

**Table 12-9. Settings for Lookup Table Input**

| Setting Value | Input Vector Pair |
|---|---|
| `GL_LIGHT_ENV_NH_DMP` | The normal and half vectors (default). |
| `GL_LIGHT_ENV_VH_DMP` | The view and half vectors. |
| `GL_LIGHT_ENV_NV_DMP` | The normal and view vectors. |
| `GL_LIGHT_ENV_LN_DMP` | The light and normal vectors. |
| `GL_LIGHT_ENV_SP_DMP` | The inverse light vector and the spotlight vector (cannot be used with RR, RG, RB, and FR). |
| `GL_LIGHT_ENV_CP_DMP` | The tangent vector and the projection of the half vector onto the tangent plane (cannot be used with RR, RG, RB, and FR). |

## 12.2.12. Distance Attenuation of Light

The following equation calculates input values for the distance attenuation of light.

$$Input_{DistAtt} = Scale \times \sqrt{\left(f_{position} - l_{position}\right)^2} + Bias$$

$f_{position}$ is the fragment position and $l_{position}$ is the light position. Both are expected to use eye coordinates. Position only has meaning for a point light source. It means nothing for a directional light source.

This equation shows that the distance between a fragment and a light is multiplied by a scale value, and then added to a bias value to calculate an input value for the distance attenuation of light.

Create lookup tables that take input between `0.0` and `1.0` (absolute values). Use the `glUniform1f()` function to set the scale value in the reserved uniform `dmp_FragmentLightSource[i].distanceAttenuationScale` and the bias value in `dmp_FragmentLightSource[i].distanceAttenuationBias`.

When `GL_LIGHT_ENV_LAYER_CONFIG7_DMP` is specified as the layer configuration's setting value, you must disable distance attenuation. Set `dmp_FragmentLightSource[i].distanceAttenuationEnabled` to `GL_FALSE`.

## 12.2.13. Texture Combiner Settings

The primary and secondary colors calculated by fragment lighting can each be used as an input source to a texture combiner.

To use them, set the reserved uniforms for the input source (`dmp_TexEnv[i].srcRgb` and `dmp_TexEnv[i].srcAlpha`) to `GL_FRAGMENT_PRIMARY_COLOR_DMP` for the primary color, and to `GL_FRAGMENT_SECONDARY_COLOR_DMP` for the secondary color.

The output value is (0.0, 0.0, 0.0, 1.0) when lighting is disabled (when `dmp_FragmentLighting.enabled` is `GL_FALSE`).

# 12.3. Bump Mapping

Bump mapping is a feature of fragment lighting that perturbs (alters) a fragment's normal and tangent vectors according to a normal map that is input as a texture. Bump mapping can make an object appear to have shadows caused by surface irregularities. This allows you to render a simple model that looks complex but actually has a small polygon count.

## 12.3.1. Reserved Uniform

The following reserved uniforms are used for bump mapping.

### Normal Maps

For the normal map texture for bump mapping (`dmp_LightEnv.bumpSelector`), specify the texture unit to which the texture is bound. A normal map texture is created with the x, y, and z components of the perturbation vectors encoded in the R, G, and B components, respectively. A vector value of `–1.0` is encoded as the minimum luminance (`0` in an 8-bit format), and `1.0` is encoded as the maximum luminance (255 in an 8-bit format).

### Perturbation Mode

To enable bump mapping, set the perturbation mode (`dmp_LightEnv.bumpMode`) to any value other than `GL_LIGHT_ENV_BUMP_NOT_USED_BUMP`. The following table shows the perturbation modes.

**Table 12-10. Perturbation Modes**

| Perturbation Mode | Perturbed Vectors |
|---|---|
| `GL_LIGHT_ENV_BUMP_NOT_USED_BUMP` | None. |
| `GL_LIGHT_ENV_BUMP_AS_BUMP_DMP` | Normal vectors (bump mapping). |
| `GL_LIGHT_ENV_BUMP_AS_TANG_DMP` | Tangent vectors (tangent mapping). |

### Normal Recalculation

If `dmp_LightEnv.bumpRenorm` is set to `GL_TRUE` to enable recalculation of normal vectors, the z-component of the normal vector is not obtained from the B component sampled from a texture. Instead, the z-component is recalculated from the x-component and the y-component.

$b_z = \sqrt{1.0 - \left(b_x^2 + b_y^2\right)}$ **Note:** If the expression inside the square root is negative, the result is 0.

In most cases, recalculating values yields better results than sampling them from a texture. This recalculation feature must be enabled, if bump mapping (of normals) uses a texture that only has R and G components (GL_HILO8_DMP). However, if you have selected tangent mapping as the perturbation mode for use with a technique such as anisotropic reflections, we recommend that you avoid using this feature. This is because tangent mapping (for fragment lighting) expects input perturbation tangents that do not have a z-component. If the recalculation feature is enabled, nonzero values may be generated in the z-component.

If recalculation is disabled, the perturbation normal vectors sampled from the texture are not normalized before they are used. Make sure that you normalize values before storing them in a texture. Re-enable normal calculation when point sampling (GL_NEAREST) is not configured as the texture filter mode because filtering can cause the non-normalized values to be used as the perturbation normals.

**Table 12-11. Reserved Uniforms Used for Bump Mapping**

| Reserved Uniform | Type | Setting Value |
|---|---|---|
| dmp_LightEnv.bumpSelector | int | Specifies the texture unit to use for the normal map.<br>GL_TEXTURE0 (default)<br>GL_TEXTURE1<br>GL_TEXTURE2<br>GL_TEXTURE3 |
| dmp_LightEnv.bumpMode | int | Specifies the perturbation mode for normal and tangent vectors.<br>GL_LIGHT_ENV_BUMP_NOT_USED_DMP: No bump mapping (default).<br>GL_LIGHT_ENV_BUMP_AS_BUMP_DMP: Perturb normal vectors.<br>GL_LIGHT_ENV_BUMP_AS_TANG_DMP: Perturb tangent vectors. |
| dmp_LightEnv.bumpRenorm | bool | Specifies whether to regenerate the third component of the normal vector.<br>GL_TRUE: Regenerate.<br>GL_FALSE: Do not regenerate (default). |

## 12.4. Shadows

3DS shadows are rendered in two passes. First, the shadow accumulation pass creates a shadow buffer (that contains the scene's depth values taking the light source as the origin), which is then referenced by the shadow lookup pass to cast shadows. The shadow intensity information collected, along with the depth values in the first pass, allows you to represent soft shadows.

### 12.4.1. Shadow Accumulation Pass

The shadow accumulation pass requires that the fragment operation mode (dmp_FragOperation.mode) be switched to shadow mode (GL_FRAGOP_MODE_SHADOW_DMP), and that the shadow information (depth values and shadow intensity) be stored in a shadow texture (with a format of GL_SHADOW_DMP and a type of GL_UNSIGNED_INT). Note that only texture unit 0

(GL_TEXTURE0) can write shadow information to a shadow texture. Also note that mipmaps cannot be applied to shadow textures.

When the fragment pipeline switches to shadow mode, shadow information is output to the attachment point for the color buffer rather than the depth or stencil buffer. As a result, a shadow texture must be attached to the color buffer's attachment point (GL_COLOR_ATTACHMENT0). Render targets attached to depth and stencil attachment points are ignored in shadow mode. The alpha and stencil tests are skipped.

You can use the following procedure to create a shadow texture and specify a render target.

**Code 12-6. Creating a Shadow Texture and Specifying a Render Target**

```
glActiveTexture(GL_TEXTURE0);
glGenTextures(1, &shadowTexID);
glBindTexture(GL_TEXTURE_2D, shadowTexID);
glTexImage2D(GL_TEXTURE_2D, 0, GL_SHADOW_DMP, shadowWidth, shadowHeight, 0,
        GL_SHADOW_DMP, GL_UNSIGNED_INT, 0);
glGenFramebuffers(1, &shadowFboID);
glBindFramebuffer(GL_FRAMEBUFFER, shadowFboID);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D,
        shadowTexID, 0);
```

Shadow information is accumulated using the coordinate system of the light source. Shadow information comprises the depth from the light source (the depth values) and the shadow intensity. When rendering shadows, there are no shadows wherever the G component of the color information is 1.0 (the R, B, and A components have no effect), there are opaque hard shadows wherever the G component of the color information is 0.0, and there are non-opaque shadows (soft shadows) everywhere else.

When opaque hard shadows are rendered, only the shadow depth values are updated. The shadow intensity does not change. The depth of a fragment and its corresponding pixel in the shadow buffer are compared (using GL_LESS). If the fragment has a smaller value, the depth value in the shadow buffer is updated.

When non-opaque soft shadows are rendered, only the shadow intensity is updated. The shadow depth values do not change. The depth of a fragment and its corresponding pixel in the shadow buffer are compared (using GL_LESS). If the fragment has a smaller value, the shadow intensity is also compared (using GL_LESS) and then, if the fragment still has a smaller value, the shadow intensity in the shadow buffer is updated.

When you initialize the color buffer in the shadow accumulation pass you must set the clear color to (1.0, 1.0, 1.0, 1.0) by using the glClearColor() function and specify GL_COLOR_BUFFER_BIT to the glClear() function. Note that you must set all color components (R, G, B, and A)—not just the G component—equal to 1.0 in the clear color.

The next shadow lookup pass is processed in eye coordinates, so the depth values must be created by using linear interpolation in eye space. (In most cases this differs from OpenGL, which uses non-linear relationships.) As a result, you must set a value in the reserved uniform for the w-buffer's scale factor (dmp_FragOperation.wScale), using the glUniform1f() function. This has an initial value of 0.0, which results in the same non-linear relationship as OpenGL. Depth values have a lower valid precision around the far clipping plane, when the near clipping plane is close to the viewpoint. To use linear interpolation, with f set as the clip value for the far clipping plane, set the scale factor to 1.0/f for a perspective projection or to 1.0 for an orthographic projection.

An object casts a hard shadow if it is rendered with a G color component of 0.0. This is generally implemented by disabling textures, and then implementing a vertex shader that outputs those vertex colors with G components of 0.0. Every other rendered color is treated as a soft shadow.

Several rendering passes may be necessary to accumulate the required shadow information. Information for non-opaque shadows must be accumulated after information for opaque shadows. Results are not guaranteed if information is accumulated in the opposite order or in alternating

order. When light sources do not move, you can generate shadow textures more efficiently by rendering motionless objects alone to a shadow texture ahead of time. Simplifying object shapes and decreasing polygon counts are also effective ways to improve performance.

Texture unit 0 and the texture combiners are configured as follows, when shadows are rendered using a vertex shader implementation that outputs vertex colors.

**Code 12-7. Sample Settings for Texture Units and Texture Combiners When Rendering Shadows**

```
glUniform1i(glGetUniformLocation(progID, "dmp_Texture[0].samplerType"),
        GL_FALSE);
glUniform1i(glGetUniformLocation(progID, "dmp_TexEnv[0].combineRgb"),
        GL_REPLACE);
glUniform1i(glGetUniformLocation(progID, "dmp_TexEnv[0].combineAlpha"),
        GL_REPLACE);
glUniform3i(glGetUniformLocation(progID, "dmp_TexEnv[0].operandRgb"),
        GL_SRC_COLOR, GL_SRC_COLOR, GL_SRC_COLOR);
glUniform3i(glGetUniformLocation(progID, "dmp_TexEnv[0].operandAlpha"),
        GL_SRC_ALPHA, GL_SRC_ALPHA, GL_SRC_ALPHA);
glUniform3i(glGetUniformLocation(progID, "dmp_TexEnv[0].srcRgb"),
        GL_PRIMARY_COLOR, GL_PRIMARY_COLOR, GL_PRIMARY_COLOR);
glUniform3i(glGetUniformLocation(progID, "dmp_TexEnv[0].srcAlpha"),
        GL_PRIMARY_COLOR, GL_PRIMARY_COLOR, GL_PRIMARY_COLOR);
```

## 12.4.2. Shadow Lookup Pass

The shadow lookup pass requires that the fragment operation mode be switched to normal mode (`GL_FRAGOP_MODE_GL_DMP`), and that texture unit 0 (`GL_TEXTURE0`) be configured to reference the shadow texture that has accumulated the shadow information. To reference a shadow texture, you must set the reserved uniform `dmp_TexEnv[0].samplerType` to `GL_TEXTURE_SHADOW_2D_DMP` (which specifies shadow textures), and bind the texture that has accumulated shadow information to `GL_TEXTURE_2D`. When fragment lighting is enabled, its primary and secondary colors are used as texture combiner input. Otherwise, vertex colors and output from texture unit 0 are used.

The reserved uniform `dmp_Texture[0].perspectiveShadow` indicates whether a perspective projection or orthographic projection was applied when the shadow accumulation pass was run. Specify `GL_TRUE` for perspective projection or `GL_FALSE` for orthographic projection.

**Code 12-8. Sample Settings When Fragment Lighting Is Disabled**

```
glUniform1i(glGetUniformLocation(progID, "dmp_Texture[0].samplerType"),
        GL_TEXTURE_SHADOW_2D_DMP);
glUniform1i(glGetUniformLocation(progID, "dmp_Texture[0].perspectiveShadow"),
        GL_TRUE);
glUniform1i(glGetUniformLocation(progID, "dmp_TexEnv[0].combineRgb"),
        GL_MODULATE);
glUniform1i(glGetUniformLocation(progID, "dmp_TexEnv[0].combineAlpha"),
        GL_MODULATE);
glUniform3i(glGetUniformLocation(progID, "dmp_TexEnv[0].operandRgb"),
        GL_SRC_COLOR, GL_SRC_COLOR, GL_SRC_COLOR);
glUniform3i(glGetUniformLocation(progID, "dmp_TexEnv[0].operandAlpha"),
        GL_SRC_ALPHA, GL_SRC_ALPHA, GL_SRC_ALPHA);
glUniform3i(glGetUniformLocation(progID, "dmp_TexEnv[0].srcRgb"),
        GL_TEXTURE0, GL_PRIMARY_COLOR, GL_PRIMARY_COLOR);
glUniform3i(glGetUniformLocation(progID, "dmp_TexEnv[0].srcAlpha"),
        GL_TEXTURE0, GL_PRIMARY_COLOR, GL_PRIMARY_COLOR);
glUniform1i(glGetUniformLocation(progID, "dmp_FragOperation.mode"),
        GL_FRAGOP_MODE_GL_DMP);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, shadowTexID);
```

Both 3DS and OpenGL reference shadow information from textures. The texture unit compares texture coordinates and shadow information obtained from texels. For this to work properly, the

correct texture coordinates must be specified. Note that a shadow texture is referenced using texture coordinates (s/q, t/q, r/q) in OpenGL, and (s/r, t/r, r - bias) on 3DS. The texture transformation matrix has already been applied to texture coordinates (s, t, r, q), and the reserved uniform `dmp_Texture[0].shadowZBias` specifies the bias value. If shadow information is accumulated by an orthographic projection, texture coordinates s and t are referenced directly. A perspective projection, however, requires adjustments to the texture transformation matrix and bias value.

The following texture transformation matrix and bias value could be used to compare the texture coordinates and shadow information accumulated by a perspective projection.

### Texture Transformation Matrix and Bias for Perspective Projection

$$texture\_matrix = \begin{bmatrix} \frac{-1}{f-n}\frac{-n}{2r} & 0 & \frac{-1}{f-n}\frac{1}{2} & 0 \\ 0 & \frac{-1}{f-n}\frac{-n}{2t} & \frac{-1}{f-n}\frac{1}{2} & 0 \\ 0 & 0 & \frac{-1}{f-n} & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$bias = \frac{n}{f-n}$$

### Texture Transformation Matrix and Bias for Parallel Projection

$$texture\_matrix = \begin{bmatrix} \frac{1}{2r} & 0 & 0 & \frac{1}{2} \\ 0 & \frac{1}{2t} & 0 & \frac{1}{2} \\ 0 & 0 & \frac{-1}{f-n} & \frac{-n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$bias = 0$$

*n* is the clip value at the near clipping plane, *f* is the clip value at the far clipping plane, and *r* and *t* are the right and top side values of the frustum.

When calculating the value that is compared to the shadow buffer depth (that is, *r* – *bias* ), if the texture coordinate *r* is not within the range from `0.0` to `1.0`, *r* is clamped to between `0.0` and `1.0` before *bias* is subtracted. To compare values correctly, specify a *bias* of `0` for any objects placed beyond the far clipping plane in the coordinate system of the light source used during the shadow accumulation pass.

You can create a texture transformation matrix with the following OpenGL code.

**Code 12-9. Creating a Texture Transformation Matrix With OpenGL Code**

```
glMatrixMode(GL_TEXTURE);
glLoadIdentity();
// (glFrustum(-r, r, -t, t, n, f))
glFrustumf(r/n, -3r/n, t/n, -3t/n, 1.0f, 0.0f);
glScalef(-1.0f/(f-n), -1.0f/(f-n), -1.0f/(f-n));
// (glOrtho(-r, r, -t, t, n, f))
glOrthof(-3r, r, -3t, t, 2n-f, f);
```

By setting the border color and texture wrapping mode, you can control the sampling results for texture coordinates that are less than `0.0` or greater than `1.0`. The texture wrapping mode

guarantees that when GL_CLAMP_TO_BORDER is configured for the s and t texture coordinates, out-of-range sampling values are set to the border color (which must have a value of 0.0 or 1.0 for all components). Sampling results are undefined in the current implementation if the wrapping mode is not GL_CLAMP_TO_BORDER or if the border color is neither (0.0, 0.0, 0.0, 0.0) nor (1.0, 1.0, 1.0, 1.0).

**Code 12-10. Sample Settings for the Wrapping Mode and Border Color**

```
glBindTexture(GL_TEXTURE_2D, shadowTexID);
GLfloat bcolor[] = {1.0f, 1.0f, 1.0f, 1.0f};
glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, bcolor);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_LOD, 0);
```

If the depth values in light source coordinates are greater than the depth values in shadow texels, the shadow intensity will be 0.0 after the comparison. Otherwise, the shadow intensity in the shadow texels is used. Texture units output the shadow intensity as a value for each RGBA component.

## 12.4.3. Omnidirectional Shadow Mapping

You can combine cube mapping with shadow textures to implement omnidirectional shadow mapping.

This implementation requires that you bind a cube map texture (one of six types: GL_TEXTURE_CUBE_MAP_POSITIVE_{X,Y,Z} or GL_TEXTURE_CUBE_MAP_NEGATIVE_{X,Y,Z}) to the texture unit during the shadow accumulation pass, render six times, and then specify GL_TEXTURE_SHADOW_CUBE_DMP as the texture to reference during the shadow lookup pass (dmp_Texture[0].samplerType). Also, you must specify GL_CLAMP_TO_EDGE as the texture coordinate wrapping mode to use in both the S and T directions.

## 12.4.4. Soft Shadows Using the Silhouette Shader

Shadow textures contain depth values and shadow intensity. The sampling value output by a texture unit that references a shadow texture is either the color black (0.0, 0.0, 0.0), within a shadow region, or the shadow intensity saved in the shadow texture (between 0.0 and 1.0), outside of a shadow region. You can represent soft shadows by setting this shadow intensity to the appropriate values.

The 3DS silhouette shader can render silhouette primitives (edges) as soft shadow regions, whose shadow intensity changes gradually until they disappear. Opaque hard shadows are rendered before the silhouette shader is used for rendering. Set the G component of the color at the silhouette edge to 1.0, and set the G component of the vertex color output by the vertex shader to 0.0. All other settings are the same as those used to render opaque hard shadows. This renders the rectangle for a silhouette edge so that its G component gradually changes from 0.0 on the object side to 1.0 on the outer side.

During the shadow accumulation pass for these soft shadow regions, you can apply additive modulation to the shadow intensity, according to the relative distance to an object. This is called an *attenuation factor*, and it can adjust the width of the soft shadow region through the reserved uniforms for the soft shadow bias (dmp_FragOperation.penumbraBias) and scale (dmp_FragOperation.penumbraScale). By adjusting these values, you can represent more

natural soft shadows that narrow close to the object.

The attenuation factor is calculated by the following equation. The equation uses $Z_{frag}$ to represent a fragment's depth value and $Z_{rec}$ to represent the depth value stored in a shadow texture. The shadow intensity is not attenuated properly if objects have not already been rendered and depth values have not already been saved in a shadow texture.

$$\frac{1}{bias + scale \times \frac{1 - Z_{frag}}{Z_{rec}}}$$

## 12.4.5. Handling Shadow Artifacts

### 12.4.5.1. Self-Shadow Aliasing

Various problems can occur during multiple-pass shadow rendering. For example, by accidentally casting a shadow on itself (called *self-shadow aliasing*) a fragment can cause a moiré pattern to be rendered. This occurs when depth values from the shadow accumulation pass are slightly smaller than depth values from the shadow lookup pass (measured from the light source). Unnatural shadows such as these are called *shadow artifacts*.

One way to suppress shadow artifacts is to apply a negative offset (bias) to depth values during the shadow lookup pass. You can set the bias value in the reserved uniform `dmp_Texture[0].shadowZBias`.

**Code 12-11. Sample Settings to Suppress Self-Shadow Aliasing**

```
glUniform1f(glGetUniformLocation(progID, "dmp_Texture[0].shadowZBias"),
            1.2f*n/(f-n));
```

### 12.4.5.2. Silhouette Shadow Artifacts

As described previously, the shadow intensity in a shadow texture is output as a sampling value for any location that is determined to be outside of a shadow region during the shadow lookup pass. Although non-shadow regions usually have a shadow intensity of `1.0` and no brightness attenuation, the soft shadow regions rendered by the silhouette shader can have an output shadow intensity that is not `1.0`, and thereby have brightness attenuation. This causes shadow artifacts to occur for some objects.

The method used to suppress self-shadow aliasing does not work for silhouette shadow artifacts, but these artifacts can still be suppressed through adjustments to shadow settings and texture combiner settings.

Because the main cause of these artifacts is the brightness attenuation of a light source by soft shadows on a parallel plane, the shadow texture output (shadow attenuation) can be calculated by using the following equation.

*ShadowAttenuation* = 1.0 – *f* (1.0 – *ShadowIntensity*)

Where *f* is a function that returns a value close to `0.0` when the fragment normals are perpendicular to the light source and `1.0` when the fragment normals are parallel to the

light source.

This equation yields a shadow attenuation of `1.0` (shadows have no effect) when a fragment's normals are nearly perpendicular to the light source and yields the shadow intensity itself when the normals are nearly parallel to the light source.

This shadow attenuation term can be implemented by using the Fresnel factor (lookup table FR) as the *f* function, reserved uniforms for the lighting environment (`dmp_LightEnv.shadowAlpha` and `dmp_LightEnv.invertShadow`) as the inverse of the shadow intensity, and texture combiner settings as the inverse of the final value.

Lighting is configured as follows. Note that the Fresnel factor affects only the alpha component. The alpha component is multiplied by the texture combiners. The *f* function is configured to return the square of the input value.

**Code 12-12. Sample (Lighting) Settings to Suppress Silhouette Shadow Artifacts**

```
glUniform1i(glGetUniformLocation(progID, "dmp_FragmentLighting.enabled"),
        GL_TRUE);
// ..other code..
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutInputFR"),
        GL_LIGHT_ENV_LN_DMP);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.config"),
        GL_LIGHT_ENV_LAYER_CONFIG1_DMP);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.fresnelSelector"),
        GL_LIGHT_ENV_PRI_SEC_ALPHA_FRESNEL_DMP);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.shadowAlpha"),
        GL_TRUE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.invertShadow"),
        GL_TRUE);

GLfloat lut[512];
int j;
memset(lut, 0, sizeof(lut));
for (j = 1; j < 128; j++)
{
    lut[j] = powf((float)j/127.0f, 2.0f);
    lut[j+255] = lut[j] - lut[j-1];
}
glTexImage1D(GL_LUT_TEXTURE0_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
        GL_LUMINANCEF_DMP, GL_FLOAT, lut);
glUniform1i(glGetUniformLocation(progID, "dmp_FragmentMaterial.samplerFR"), 0);
```

In this code, *f* (`1.0` – *ShadowIntensity*) is output as the fragment's primary and secondary alpha component.

The final shadow attenuation factor is calculated next from the texture combiner settings and is multiplied by the fragment's primary color. Note that the fragment's primary alpha value is inverted here (`GL_ONE_MINUS_SRC_ALPHA`).

**Code 12-13. Sample (Texture Combiner) Settings to Suppress Silhouette Shadow Artifacts**

```
glUniform1i(glGetUniformLocation(progID, ("dmp_TexEnv[0].combineRgb"),
        GL_MODULATE);
glUniform1i(glGetUniformLocation(progID, ("dmp_TexEnv[0].combineAlpha"),
        GL_REPLACE);
glUniform3i(glGetUniformLocation(progID, ("dmp_TexEnv[0].operandRgb"),
        GL_SRC_COLOR, GL_ONE_MINUS_SRC_ALPHA, GL_SRC_COLOR);
glUniform3i(glGetUniformLocation(progID, ("dmp_TexEnv[0].operandAlpha"),
        GL_SRC_ALPHA, GL_SRC_ALPHA, GL_SRC_ALPHA);
glUniform3i(glGetUniformLocation(progID, ("dmp_TexEnv[0].srcRgb"),
        GL_FRAGMENT_PRIMARY_COLOR_DMP, GL_FRAGMENT_PRIMARY_COLOR_DMP,
        GL_PRIMARY_COLOR);
glUniform3i(glGetUniformLocation(progID, ("dmp_TexEnv[0].srcAlpha"),
        GL_PRIMARY_COLOR, GL_PRIMARY_COLOR, GL_PRIMARY_COLOR);
```

## 12.4.6. Reserved Uniform

The following table lists the reserved uniforms used for shadows.

**Table 12-12. Reserved Uniforms Used for Shadows**

| Reserved Uniform | Type | Setting Value |
|---|---|---|
| `dmp_Texture[0].samplerType` | `int` | Specifies the type of texture to reference. The following two types can be used for shadows.<br>`GL_TEXTURE_SHADOW_2D_DMP`<br>`GL_TEXTURE_SHADOW_CUBE_DMP` |
| `dmp_Texture[0].perspectiveShadow` | `bool` | Specifies whether a perspective projection has been applied as the projection transformation.<br>`GL_TRUE`: Has been applied (default).<br>`GL_FALSE`: Has not been applied. |
| `dmp_Texture[0].shadowZBias` | `float` | Specifies the bias value for the negative offset to apply to depth values during the lookup pass.<br>0.0 (default) |
| `dmp_FragOperation.mode` | `int` | Specifies the fragment operation mode. Specifies shadow mode during the accumulation pass and standard mode during the lookup pass.<br>`GL_FRAGOP_MODE_GL_DMP` (standard mode)<br>`GL_FRAGOP_MODE_SHADOW_DMP` (shadow mode) |
| `dmp_FragOperation.wScale` | `float` | Specifies the scale factor for the w-buffer.<br>0.0 (default) |
| `dmp_FragOperation.penumbraScale` | `float` | Specifies the scale value for soft shadows.<br>1.0 (default) |
| `dmp_FragOperation.penumbraBias` | `float` | Specifies the bias value for soft shadows.<br>0.0 (default) |

## 12.4.7. Checking Shadow Texture Content

To check the image that was rendered to a shadow texture, attach the shadow texture to the current color buffer, and then read the texel data by calling the `glReadPixels()` function and specifying `GL_RGBA` for the *`format`* parameter and `GL_UNSIGNED_BYTE` for the *`type`* parameter. When accessed through a u32 pointer, the least-significant 8 bits of texel data store the shadow intensity, and the most-significant 24 bits store the depth value. Shift this to the right by 8 bits to get only the depth value.

Because shadow textures store an 8-bit shadow intensity and a 24-bit depth value, and both of these represent values in the range from `0.0` to `1.0`, the shadow intensity and depth value have different precisions.

The shadow intensity takes a value between `0x00` and `0xFF`, where `0xFF` indicates the absence of a soft shadow region and all other values indicate the presence of a soft shadow region. In other words, a value of `0x00` indicates a shadow intensity of 0.0 and `0xFF` indicates a shadow intensity of 1.0.

The depth value is scaled so that `0x000000` is the near value and `0xFFFFFF` is the far value. In other words, a value of `0x000000` indicates a depth value of 0.0 and `0xFFFFFF` indicates a depth value of 1.0. Note that this scaling is uniform if the w buffer is enabled during the shadow accumulation pass. For more information, see 10.3.3. W-Buffer.

A shadow intensity of `0xFF` and a depth value of `0xFFFFFF` are written to the shadow texture wherever there are no shadows. A shadow intensity of `0xFF` and a depth value other than `0xFFFFFF` are written to the shadow texture wherever there are only hard shadows. A shadow intensity other than `0xFF` and a depth value other than `0xFFFFFF` are used in regions that have both hard and soft shadows.

**Table 12-13. Types of Shadows and the Values Written to Shadow Textures**

| Shadow Type | Shadow Intensity | Depth Values |
|---|---|---|
| No shadows | 0xFF | 0xFFFFFF |
| Only hard shadows | 0xFF | Value other than 0xFFFFFF |
| Soft shadows | Value other than 0xFF | Value other than 0xFFFFFF |

# 12.5. Fog

Although 3DS fog has nearly the same features as fog defined in OpenGL ES 1.1, the effect of 3DS fog is determined by projection-corrected depth values, whereas the effect of OpenGL fog is determined by the distance from the viewpoint. Another difference is that fog coefficients are specified by lookup tables. There are also no settings for fog properties, such as the beginning, end, and density.

As in OpenGL, the following equation determines a fragment's color after fog has been applied.

$$Color = f \times C_{fragment} + (1 - f) \times C_{fog}$$

$f$ is the fog coefficient (between `0.0` and `1.0`).

## 12.5.1. Reserved Uniform

The following reserved uniforms are used for fog.

### Fog Mode

To enable fog, specify a fog mode (`dmp_Fog.mode`) of `GL_FOG` using the `glUniform1i()` function. To disable it, specify `GL_FALSE`.

### Fog Color

Use the `glUniform3f()` function to set the fog color (`dmp_Fog.color`). Only the RGB components are set. (The alpha component is not.)

### Fog Coefficients

Fog coefficients configure a lookup table that takes depth values in window coordinates as input. Call the `glUniform1i()` function to specify the reserved uniform (`dmp_Fog.sampler`) to the lookup table number to use. Note that the specified lookup table number `GL_LUT_TEXTURE`*i*`_DMP`, where *i* represents a number from 0 to 31, does not specify the name (ID) of the texture nor `GL_LUT_TEXTURE`*i*`_DMP` directly.

### Whether to Invert Input Depth Value

You can choose whether to invert input values (changing $z$ to $1 - z$) for the fog coefficient lookup table. To invert values, set the reserved uniform `dmp_Fog.zFlip` to `GL_TRUE`, using the `glUniform1i()` function.

**Table 12-14. Reserved Uniforms Used for Fog**

| Reserved Uniform | Type | Setting Value |
|---|---|---|
| `dmp_Fog.mode` | `int` | Specifies the mode for processing the fog pipeline (the fog mode).<br>`GL_FOG`<br>`GL_GAS_DMPGL_FALSE` (default) |
| `dmp_Fog.color` | `vec3` | Specifies the fog color. There is no alpha component.<br>Each component has a value between 0.0 and 1.0.<br>(0.0, 0.0, 0.0) by default. |
| `dmp_Fog.zFlip` | `bool` | Specifies whether to invert the depth values used as input to the lookup table for fog coefficients.<br>`GL_TRUE` or `GL_FALSE` (default). |
| `dmp_Fog.sampler` | `int` | Specifies the lookup table to use for fog coefficients.<br>0 through 31 |

## 12.5.2. Creating and Specifying Lookup Tables

A lookup table for fog coefficients takes input values between `0.0` and `1.0`, and has a fixed *`width`* of 256. Like other lookup tables, it stores the output values in the first 128 elements and the differences between output values in the last 128 elements.

You must be careful about how you convert input values, when you create lookup tables to implement OpenGL fog coefficients on a 3DS system. The input depth values are specified in window coordinates, with the near clipping plane at the minimum value (`0.0`) and the far clipping plane at the maximum value (`1.0`). For a perspective projection, however, these depth values have a non-linear relationship with depth values in eye coordinates. As a result, lookup table output must be calculated using depth values that were converted from window coordinates into eye coordinates.

Considering that input in window coordinates (between `0.0` and `1.0`) is mapped to clip coordinates (between `0.0` and `−1.0`), the following equation is used to convert values into eye coordinates. Note that the sign of the input is reversed.

$$(Xe\ Ye\ Ze\ We) = (0.0\ 0.0\ -Zw\ 1.0) \times M_{projection}^{-1}$$

Fog coefficients are a function of the distance to a fragment from the origin in eye coordinates. Input to this function can be approximated as the distance $-Ze / We$ between the xy plane and the fragment in eye coordinates.

This is shown by the following sample code. `FogCoef` is the fog coefficient function.

**Code 12-14. Creating a Fog Lookup Table**

```
float Fog_LUT[256], Fog_c[128 + 1];
int i;
Matrix44 invPM;
Vector4 v_eye, v_clip(0.0f, 0.0f, 0.0f, 1.0f);

MTX44Inverse(&invPM, &projMatrix);
Vector4 v0(invPM.m[0]);
Vector4 v1(invPM.m[1]);
Vector4 v2(invPM.m[2]);
Vector4 v3(invPM.m[3]);
for (i = 0; i <= 128; i++) {
    v_clip.z = -(static_cast<f32>(i)) / 128;
    v_eye.x = VEC4Dot(&v0, &v_clip);    v_eye.y = VEC4Dot(&v1, &v_clip);
    v_eye.z = VEC4Dot(&v2, &v_clip);    v_eye.w = VEC4Dot(&v3, &v_clip);
    Fog_c[i] = -(v_eye.z / v_eye.w);
}
for (i = 0; i < 128; i++) {
    Fog_LUT[i] = FogCoef(Fog_c[i]);
    Fog_LUT[128 + i]= FogCoef(Fog_c[i + 1]) - FogCoef(Fog_c[i]);
}
```

**Note:** OpenGL fog is affected by the eye coordinates' z value, but 3DS fog is affected by depth values that have been corrected with a perspective projection. As a result, fog changes when the near and far clipping planes change, and the same lookup table can produce different effects depending on whether the w-buffer or a normal depth buffer is used. For more information, see 10.3.3. W-Buffer.

# 12.6. Gas Rendering

Gas rendering uses the fog feature, configured in gas mode, to render gaseous bodies from density information, which is itself generated based on depth values of polygon objects. Gaseous bodies require three rendering passes: the polygon object rendering pass, which generates depth values for the polygon objects; the density rendering pass, which accumulates density information in a gas texture; and the shading pass, which renders gaseous bodies by referencing a gas texture.

## 12.6.1. Polygon Object Rendering Pass

This pass renders polygon objects as usual, generating depth information that is used to determine where the polygon objects and gaseous bodies intersect. The content of the depth buffer is the only rendering result that is used by the next pass.

## 12.6.2. Density Rendering Pass

This pass uses a means such as point sprites to render the smallest units comprising a gaseous body and gas particles (a texture with a density pattern), and it accumulates a gas's depth information in the color buffer. The content of the color buffer is copied to the gas texture and is used for the next path.

### Preparing Color Buffers and Gas Textures

Prepare a color buffer with an internal format of `GL_GAS_DMP` and a texture (gas texture), both of which are used to copy the rendering results. Create the color buffer with the same size as the depth buffer, but the gas texture must have a width and height (in texels) that are both powers of 2, and `GL_UNSIGNED_SHORT` must be specified for *`type`*. Because gas textures always use point sampling, minification and magnification filters have no effect. Use `GL_NEAREST` as the filter setting. Note that mipmaps cannot be applied to gas textures.

**Code 12-15. Preparing a Color Buffer and Gas Texture for Gas Rendering**

```
// Generating Object
glGenFramebuffers(1, &gasAccFboID);
glGenTextures(1, &gasTexID);
// Renderbuffer & Framebuffer
glGenRenderbuffers(1, &gasAccColorID);
glBindRenderbuffer(GL_RENDERBUFFER, gasAccColorID);
glRenderbufferStorage(GL_RENDERBUFFER, GL_GAS_DMP, GAS_ACC_WIDTH,
        GAS_ACC_HEIGHT);
glBindFramebuffer(GL_FRAMEBUFFER, gasAccFboID);
```

```
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_RENDERBUFFER,
        gasAccColorID);
// Gaseous Texture
glBindTexture(GL_TEXTURE_2D, gasTexID);
glTexImage2D(GL_TEXTURE_2D, 0, GL_GAS_DMP, GAS_TEX_WIDTH, GAS_TEX_HEIGHT, 0,
        GL_GAS_DMP, GL_UNSIGNED_SHORT, 0);
glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEAREST);
glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_NEAREST);
```

### Rendering Gas Particles

To render density information to the color buffer, you must switch the fragment operation mode (`dmp_FragOperation.mode`) to gas mode (`GL_FRAGOP_MODE_GAS_ACC_DMP`). Two types of density information are accumulated in the color buffer: information that is simply accumulated (D1), and information that accounts for intersections with polygon offsets (D2).

Disable the depth test, the depth mask, blending, and fog to prevent the content of the depth buffer from being updated. However, do not change the comparison function that was used for the depth test, when polygon objects were rendered.

Clear only one buffer, the color buffer to which density information is rendered. The depth buffer must not be cleared.

The R component of the gas particles to be rendered (*Df*) is accumulated in the color buffer as density information. D1 and D2 are updated to D1' and D2' by the following equations. The equation for D2' depends on the comparison function for the depth test.

$$D1' = D1 + Df$$

$$DZ = (Zb - Zf) < 0.0\ ?\ 0.0 : (Zb - Zf) \times EZ$$
$$ATT = DZ > 1.0\ ?\ 1.0 : DZ$$
$$D2' = D2 + Df \times ATT \quad \cdots \text{GL\_LESS,GL\_LEQUAL}$$
$$D2' = D2 + Df \times (1.0 - ATT) \quad \cdots \text{GL\_GREATER,GL\_GEQUAL}$$
$$D2' = D2 + Df \quad \cdots \text{GL\_ALWAYS}$$
$$D2' = D2 \quad \cdots \text{GL\_NEVER}$$

*Zb* is the depth value stored in the depth buffer, and *Zf* is the depth value of the fragment.

D1 accumulates fragment (gas particle) density information unchanged. D2 accumulates fragment density information that has been multiplied by the attenuation coefficient EZ in the depth direction and also by the difference between the depth values in the depth buffer and fragment. The EZ coefficient is a floating-point number set by the `glUniform1f()` function in the reserved uniform `dmp_Gas.deltaZ`.

#### Code 12-16. Rendering Gas Particles

```
// Change to gas accumulation mode.
glBindFramebuffer(GL_FRAMEBUFFER, gasAccFboID);
glUniform1i(glGetUniformLocation(progID, "dmp_FragOperation.mode"),
        GL_FRAGOP_MODE_GAS_ACC_DMP);
glDisable(GL_DEPTH_TEST);
glDepthMask(GL_FALSE);
glDisable(GL_BLEND);
glUniform1i(glGetUniformLocation(progID, "dmp_Fog.mode"), GL_FALSE);
// Colorbuffer Clear
glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
glClear(GL_COLOR_BUFFER_BIT);
// Set dmp_Gas.deltaZ.
glDepthFunc(GL_LESS);
glUniform1f(glGetUniformLocation(progID, "dmp_Gas.deltaZ"), 50.0f);
```

### Copying Information to a Gas Texture

After all gas particles have been rendered, the density information accumulated in the color buffer is copied to a gas texture.

The gas texture is not usually allocated with the same size as the color buffer (the width and height must be powers of 2), so the `glCopyTexSubImage2D()` function partially copies color buffer data. The gas texture is required for the next pass.

**Code 12-17.Coping Information to a Gas Texture**

```
// Bind and copy to gaseous texture.
glBindTexture(GL_TEXTURE_2D, gasTexID);
glCopyTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, 0, 0, GAS_ACC_WIDTH,
                    GAS_ACC_HEIGHT);
```

## 12.6.3. Shading Pass

This pass shades gaseous bodies by referencing the density information accumulated in a gas texture, and then blends the shading results with the color buffer contents from the polygon object rendering pass.

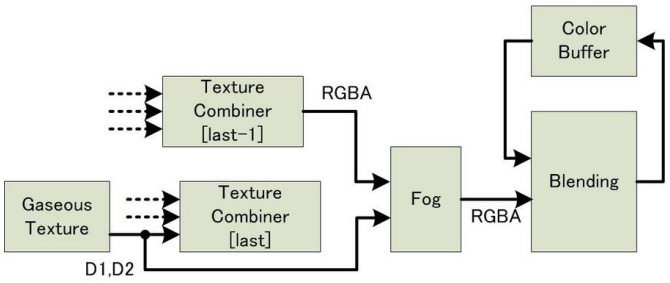
Gaseous bodies are shaded using the fog feature configured in gas mode. Special settings are required for fog input from the texture combiners.

### Fog Input (Texture Combiner Settings)

Fog takes two inputs: the RGBA components from the second-to-last texture combiner (though only the R component is actually used), and the density information from the input source, specified as input source 2 (the third component of `srcRgb` and `srcAlpha`) to the last texture combiner (the specified texture unit must sample the gas texture). Because output from the last texture combiner is ignored, the number of usable texture combiner levels is reduced by one.

Ultimately, fog output is blended with the contents of the color buffer. Because fog outputs alpha values that account for intersections with polygon objects, blending both outputs based on the fog's alpha values allows gaseous bodies to be rendered in the correct order front-to-back.

**Figure 12-5.Fog Input and Blending With the Color Buffer**

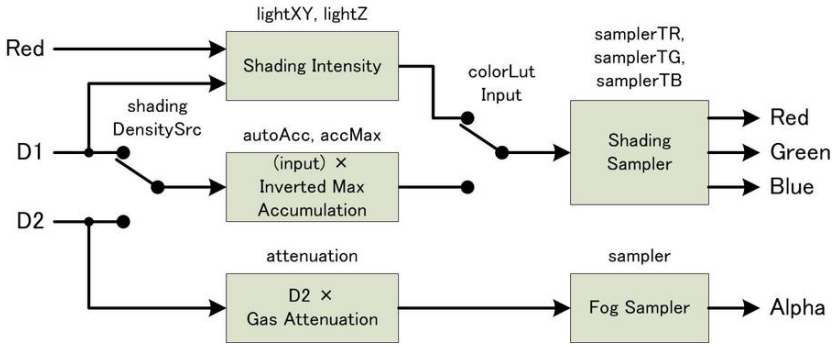



### Fog Operations in Gas Mode

In gas mode, fog shades gaseous bodies based on input information. To enable fog in the gas mode, specify a fog mode (`dmp_Fog.mode`) for `GL_GAS_DMP`.

The RGB components of the shading results are determined by shading lookup tables. To determine the alpha component, the gas attenuation value (`dmp_Gas.attenuation`) is multiplied by density information that accounts for intersections (D2), and the result is looked up in the lookup table specified by the fog coefficient (`dmp_Fog.sampler`).

**Figure 12-6. Fog in Gas Mode**




## Shading Lookup Tables

Shading lookup tables accept either density information or shading intensity as input, and provide the RGB components of the shading results as output.

Shading lookup tables are specified separately for each component using tables generated with *`width`* set to 16. Both the input and output values are between `0.0` and `1.0`. As with other lookup tables, the output values are stored in the first eight elements, the differences between output values are stored in the last eight elements, and the output values are interpolated using the delta values.

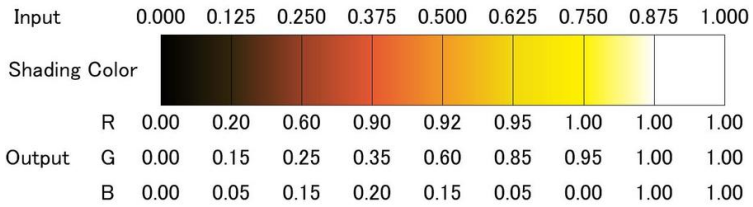
The shading lookup table to use uses the `glUniform1i()` function to specify the lookup table number as the following reserved uniform. Note that the specified lookup table number `GL_LUT_TEXTURE`*`i`*`_DMP`, where *`i`* represents a number from 0 to 31, does not specify the name (ID) of the texture nor `GL_LUT_TEXTURE`*`i`*`_DMP` directly.

**Table 12-15. Reserved Uniforms Used to Specify Shading Lookup Tables**

| Reserved Uniform | Type | Setting Value |
|---|---|---|
| `dmp_Gas.samplerTR`<br>`dmp_Gas.samplerTG`<br>`dmp_Gas.samplerTB` | `int` | Specifies the lookup table number to use as the shading lookup tables (for the R, G, and B components).<br>0 through 31 |

The following are examples of shading lookup tables and their implementations.

**Figure 12-7. A Shading Lookup Table**




**Code 12-18. Creating Shading Lookup Tables**

```
// Define
GLfloat shading_color[3 * 9] = {
    0.00f, 0.00f, 0.00f,
    0.20f, 0.15f, 0.05f,
    0.60f, 0.25f, 0.15f,
    0.90f, 0.35f, 0.20f,
    0.92f, 0.60f, 0.15f,
    0.95f, 0.85f, 0.05f,
    1.00f, 0.95f, 0.00f,
    1.00f, 1.00f, 1.00f,
```

```
    1.00f, 1.00f, 1.00f
};
GLfloat samplerTR[16], samplerTG[16], samplerTB[16];
// Table
for(int i = 0; i < 8; i++) {
    // shading color value
    samplerTR[i] = shading_color[3*i + 0];
    samplerTG[i] = shading_color[3*i + 1];
    samplerTB[i] = shading_color[3*i + 2];
    // difference of shading color value
    samplerTR[8 + i] = shading_color[3*(i + 1) + 0] - shading_color[3*i + 0];
    samplerTG[8 + i] = shading_color[3*(i + 1) + 1] - shading_color[3*i + 1];
    samplerTB[8 + i] = shading_color[3*(i + 1) + 2] - shading_color[3*i + 2];
}
// Texture
glBindTexture(GL_LUT_TEXTURE1_DMP, samplerTR_ID);
glTexImage1D(GL_LUT_TEXTURE1_DMP, 0, GL_LUMINANCEF_DMP, 16, 0,
        GL_LUMINANCEF_DMP, GL_FLOAT, samplerTR);
glBindTexture(GL_LUT_TEXTURE2_DMP, samplerTG_ID);
glTexImage1D(GL_LUT_TEXTURE2_DMP, 0, GL_LUMINANCEF_DMP, 16, 0,
        GL_LUMINANCEF_DMP, GL_FLOAT, samplerTG);
glBindTexture(GL_LUT_TEXTURE3_DMP, samplerTB_ID);
glTexImage1D(GL_LUT_TEXTURE3_DMP, 0, GL_LUMINANCEF_DMP, 16, 0,
        GL_LUMINANCEF_DMP, GL_FLOAT, samplerTB);
// Set Uniform
glUniform1i(glGetUniformLocation(progID, "dmp_Gas.samplerTR"), 1);
glUniform1i(glGetUniformLocation(progID, "dmp_Gas.samplerTG"), 2);
glUniform1i(glGetUniformLocation(progID, "dmp_Gas.samplerTB"), 3);
```

### Shading Based on Density Information

For shading that is based on density information alone (without accounting for the effect of shadows cast by light), set the reserved uniform `dmp_Gas.colorLutInput` to `GL_GAS_DENSITY_DMP`, using the `glUniform1i()` function to select density information, as the shading lookup table input.

As described for the density rendering pass, two types of density information are stored in a gas texture. One type of density information does not account for intersections with polygon objects (D1), and the other type does (D2). You can set a value in the reserved uniform `dmp_Gas.shadingDensitySrc` to select the density information to use.

Specify `GL_GAS_PLAIN_DENSITY_DMP` for D1 and `GL_GAS_DEPTH_DENSITY_DMP` for D2, using the `glUniform1i()` function. Even if you choose D1 as input to the shading lookup tables, fog in gas mode outputs alpha components calculated from density information that accounts for intersections (D2). By using alpha values while blending, polygon objects and gaseous bodies can be rendered with the correct front-to-back ordering.

Lookup table input uses values between `0.0` and `1.0`. You can multiply the density information by the reciprocal of the maximum density value to keep values in this range. The reciprocal of the maximum value can be calculated automatically. By setting the reserved uniform `dmp_Gas.autoAcc` to `GL_TRUE`, you can use the `glUniform1i()` function to calculate the reciprocal from the maximum D1 value in the density rendering pass.

When `GL_FALSE` is specified, the value set in the reserved uniform `dmp_Gas.accMax` by the `glUniform1f()` function is used as the reciprocal of the maximum value.

### Shading Based on the Shading Intensity

To use shading that is calculated from the shading intensity (accounting for the effect of shadows cast by light), set the reserved uniform `dmp_Gas.colorLutInput` to `GL_GAS_LIGHT_FACTOR_DMP`, using the `glUniform1i()` function. This selects the shading intensity as input to the shading lookup table.

The shading intensity is the total of two calculated values: the planar shading intensity (IG) and the view shading intensity (IS). IG and IS are defined as follows.

$$ig = r \times (1.0 - lightAtt \times d1)$$

*IG* = (1.0 - *ig*) × *lightMin* + *ig* × *lightMax*
*is* = *LZ* × (1.0 - *scattAtt* × *d1*)
*IS* = (1.0 - *is*) × *scattMin* + *is* × *scattMax*

*d1* is the result of multiplying the density information that does not account for intersections with polygon objects (D1), by the reciprocal of the maximum density. This mechanism is similar to the one used for shading based on density information. *r* is the R component output from the texture combiner that is used as fog input. *lightAtt*, *lightMin*, and *lightMax* are coefficients of the planar shading intensity. *LZ* is the light direction along the z-axis in eye coordinates. *scattAtt*, *scattMin*, and *scattMax* are coefficients of the view shading intensity. These all take values between `0.0` and `1.0`.

The planar shading intensity is proportional to *r* and (`1.0` – *lightAtt* × *d1*), as its equation shows. It approaches *lightMax* when *r* increases and *lightMin* when *d1* increases. Likewise, the view shading intensity is proportional to *LZ* and (`1.0` – *scattAtt* × *d1*). It approaches *scattMax* when *LZ* increases and *scattMin* when *d1* increases.

Note that the shading intensity input to a shading lookup table is larger for gaseous bodies that are less dense. This emulates the real behavior of light, which penetrates through the thin (low-density) areas of a gaseous body, and is absorbed in the thick (high-density) areas. Accordingly, *lightMin* and *scattMin* represent the shading intensity when the effect of light is small, and *lightMax* and *scattMax* represent the shading intensity when the effect of light is large. *lightAtt* and *scattAtt* set the ratio of light attenuation caused by density. Note that, depending on the values set in the shading lookup tables, *lightMin* is not necessarily less than *lightMax* (and *scattMin* is not necessarily less than *scattMax*). Also note that alpha values are determined by fog coefficients whose input is proportional to density.

Coefficients for the planar shading intensity (*lightMin*, *lightMax*, and *lightAtt*) are set, as a group, in the reserved uniform `dmp_Gas.lightXY`. Coefficients for the view shading intensity (*scattMin*, *scattMax*, and *scattAtt*), and the light direction along the z-axis in eye coordinates (LZ), are set as a group in the reserved uniform `dmp_Gas.lightZ`. The minimum value, maximum value, and attenuation are set in that order. They are followed by LZ for the view shading intensity.

The following code shows how to set the shading intensity coefficients.

**Code 12-19. Setting Shading Intensity Coefficients**

```
GLfloat lightXY[3], lightZ[4];
GLfloat lightMin, lightMax, lightAtt;
GLfloat scattMin, scattMax, scattAtt, LZ;

//...

// lightXY
lightXY[0] = lightMin;
lightXY[1] = lightMax;
lightXY[2] = lightAtt;
// lightZ
lightZ[0] = scattMin;
lightZ[1] = scattMax;
lightZ[2] = scattAtt;
lightZ[3] = LZ;
// Set Uniform
glUniform3fv(glGetUniformLocation(progID, "dmp_Gas.lightXY"), 1, lightXY);
glUniform4fv(glGetUniformLocation(progID, "dmp_Gas.lightZ"), 1, lightZ);
```

## Alpha Shading

The results of shading the alpha component are determined by output from the lookup table for fog coefficients (`dmp_Fog.sampler`), given the product of the gas attenuation (`dmp_Gas.attenuation`) and the density information that accounts for intersections (D2) as input.

Fog coefficients are normally specified using a function that approaches `0.0` when the gas density is low, and `1.0` when the gas density is high.

**Code 12-20. Setting Fog Coefficients**

```
// Fog factor
for(int i = 0; i < 128; i++) {
    fogTable[i]= 1.0f - exp(-8.0f * i / 128.0f);
}
for(int i = 0; i < 128; i++) {
    fogTable[128 + i] = fogTable[i + 1] - fogTable[i];
}
fogTable[255] = 0;
// Set LUT
glGenTextures(1, &fogLUT_ID);
glBindTexture(GL_LUT_TEXTURE0_DMP, fogLUT_ID);
glTexImage1D(GL_LUT_TEXTURE0_DMP, 0, GL_LUMINANCEF_DMP, 256, 0,
        GL_LUMINANCEF_DMP, GL_FLOAT, fogTable);
```

The following sample code sets uniforms required by the shading pass and renders a quad (polygon), to which a gas texture has been applied.

**Code 12-21. Setting Uniforms for the Shading Pass**

```
// Bind Framebuffer(Colorbuffer)
glBindFramebuffer(GL_FRAMEBUFFER, renderFboID);
// Bind Gas Texture
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, gasTexID);
glUniform1i(glGetUniformLocation(progID, "dmp_Texture[0].samplerType"),
        GL_TEXTURE_2D);
// Set TextureCombiner #5
glUniform3i(glGetUniformLocation(progID, "dmp_TexEnv[5].srcRgb"),
        GL_PREVIOUS, GL_PREVIOUS, GL_TEXTURE0);
glUniform3i(glGetUniformLocation(progID, "dmp_TexEnv[5].srcAlpha"),
        GL_PREVIOUS, GL_PREVIOUS, GL_TEXTURE0);
// Set uniform for gas shading mode.
glUniform1i(glGetUniformLocation(progID, "dmp_Fog.sampler"), 0);
glUniform1i(glGetUniformLocation(progID, "dmp_Gas.autoAcc"), GL_FALSE);
glUniform1i(glGetUniformLocation(progID, "dmp_Gas.samplerTR"), 1);
glUniform1i(glGetUniformLocation(progID, "dmp_Gas.samplerTG"), 2);
glUniform1i(glGetUniformLocation(progID, "dmp_Gas.samplerTB"), 3);
glUniform1i(glGetUniformLocation(progID, "dmp_Gas.shadingDensitySrc"),
        GL_GAS_DEPTH_DENSITY_DMP);
glUniform1i(glGetUniformLocation(progID, "dmp_Gas.colorLutInput"),
        GL_GAS_DENSITY_DMP);
glUniform1f(glGetUniformLocation(progID, "dmp_Gas.accMax"), 1.0f/6.0f);
glUniform4fv(glGetUniformLocation(progID, "dmp_Gas.lightZ"), 1, gasLightZ);
glUniform3fv(glGetUniformLocation(progID, "dmp_Gas.lightXY"), 1, gasLightXY);
// Change to gas shading mode.
glUniform1i(glGetUniformLocation(progID, "dmp_FragOperation.mode"),
        GL_FRAGOP_MODE_GL_DMP);
glUniform1i(glGetUniformLocation(progID, "dmp_Fog.mode"), GL_GAS_DMP);
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
glDisable(GL_DEPTH_TEST);
glDepthMask(GL_FALSE);
```

**Code 12-22. Rendering a Quad (Polygon) With a Gas Texture Applied**

```
// Gaseous Shading
// Texture Coord
float u0 = 0.0f;
float v0 = 0.0f;
float u1 = (GAS_ACC_WIDTH * 1.0f) / (GAS_TEX_WIDTH * 1.0f);
float v1 = (GAS_ACC_HEIGHT * 1.0f) / (GAS_TEX_HEIGHT * 1.0f);
GLfloat texCoord[8]= {u0, v0, u0, v1, u1, v1, u1, v0};
// Vertex
GLushort quadIndex[6] = {0, 1, 2, 0, 2, 3};
GLfloat vertex[16] = {
    -1.0f, -1.0f, 0.0f, 1.0f,
    -1.0f, 1.0f, 0.0f, 1.0f,
    1.0f, 1.0f, 0.0f, 1.0f,
    1.0f, -1.0f, 0.0f, 1.0f
};
// Set Array
GLfloat quadIndexID, quadVertexID, quadTexCoordID;
glGenBuffers(1, &quadIndexID);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, quadIndexID);
```

```
glBufferData(GL_ELEMENT_ARRAY_BUFFER, 6 * sizeof(GLushort), &quadIndex,
        GL_STATIC_DRAW);
glGenBuffers(1, &quadVertexID);
glBindBuffer(GL_ARRAY_BUFFER, quadVertexID);
glBufferData(GL_ARRAY_BUFFER, 16 * sizeof(GLfloat), &vertex, GL_STATIC_DRAW);
glGenBuffers(1, &quadTexCoordID);
glBindBuffer(GL_ARRAY_BUFFER, quadTexCoordID);
glBufferData(GL_ARRAY_BUFFER, 8 * sizeof(GLfloat), &texCoord, GL_STATIC_DRAW);
// Draw Quad
glEnableVertexAttribArray(0);
glEnableVertexAttribArray(1);
glBindBuffer(GL_ARRAY_BUFFER, quadVertexID);
glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 0, 0);
glBindBuffer(GL_ARRAY_BUFFER, quadTexCoordID);
glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 0, 0);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, quadIndexID);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_SHORT, 0);
```

## 12.6.4. Reserved Uniform

The following table lists the reserved uniforms used for gas.

**Table 12-16. Reserved Uniforms Used for Gas**

| Reserved Uniform | Type | Setting Value |
|---|---|---|
| dmp_FragOperation.mode | int | Specifies the fragment operation mode. Specify gas mode in the density rendering pass. Specify standard mode in any other pass.<br>GL_FRAGOP_MODE_GL_DMP (standard mode)<br>GL_FRAGOP_MODE_SHADOW_DMP (shadow mode) |
| dmp_Fog.mode | int | Specifies the mode for processing the fog pipeline. Specify gas mode in the shading pass. Disable fog in the density rendering pass.<br>Specify GL_GAS_DMP for gas mode in the shading pass, and GL_FALSE to disable fog in the density rendering pass. |
| dmp_Fog.sampler | int | Specifies the lookup table to use for fog coefficients. Fog coefficients are used to calculate alpha values for gaseous bodies.<br>0 through 31 |
| dmp_Gas.deltaZ | float | Specifies the attenuation coefficient EZ in the depth direction.<br>10.0 (default) |
| dmp_Gas.autoAcc | bool | Specifies whether to automatically calculate the reciprocal of the maximum density value.<br>GL_TRUE: Automatically calculate (default).<br>GL_FALSE: Do not automatically calculate. |
| dmp_Gas.accMax | float | Specifies the reciprocal of the maximum density value.<br>0.0 or more<br>1.0 (default) |
| dmp_Gas.samplerTR<br>dmp_Gas.samplerTG<br>dmp_Gas.samplerTB | int | Specifies the shading lookup table for each RGB component.<br>0 through 31 |
| dmp_Gas.shadingDensitySrc | int | Specifies the density information to use for shading.<br>Specify GL_GAS_PLAIN_DENSITY_DMP (default) or GL_GAS_DEPTH_DENSITY_DMP. |
| dmp_Gas.colorLutInput | int | Specifies whether the density or shading intensity is given as input to the shading lookup tables.<br>Specify GL_GAS_LIGHT_FACTOR_DMP (default) or GL_GAS_DENSITY_DMP. |
| | | Specifies factors used to control planar shading: the |

| | | |
|---|---|---|
| dmp_Gas.lightXY | vec3 | minimum intensity, maximum intensity, and density attenuation.<br>(lightMin, lightMax, lightAtt)<br>Each control value is between 0.0 and 1.0.<br>The default value is (0.0, 0.0, 0.0). |
| dmp_Gas.lightZ | vec4 | Specifies factors used to control view shading: the minimum intensity, maximum intensity, density attenuation, and effect in the view direction.<br>(scattMin, scattMax, scattAtt, LZ)<br>Each control value is between 0.0 and 1.0.<br>The default value is (0.0, 0.0, 0.0, 0.0). |
| dmp_Gas.attenuation | float | Specifies the density attenuation coefficient to use when calculating alpha values for shading.<br>0.0 or more<br>1.0 (default) |

CONFIDENTIAL

# 13. Per-Fragment Operations

Per-fragment operations update the pixel data stored at a fragment's window coordinates in the framebuffer, based on the lighting results and other related data.

This does not mean that all fragments are taken into account. Unnecessary fragments are rejected by tests, such as the alpha test, and then existing pixel data and colors are combined through blending and logical operations.

When the fragment operation mode (mp_FragOperation.mode) is in standard mode (GL_FRAGOP_MODE_GL_DMP), per-fragment operations occur as in the process in Figure 13-1.

**Figure 13-1. Per-Fragment Operations Process in Standard Mode**

The 3DS specifications differ from the OpenGL ES 2.0 specifications in the following ways.

- The scissor test is run during rasterization on a CTR system.
- The glSampleCoverage() function has not been implemented because multisampling is not supported.
- The alpha test provides features equivalent to those in OpenGL ES 1.1, but it is controlled entirely through reserved uniforms.
- The glStencil∗Separate() functions have not been implemented because the stencil test does not distinguish between front-facing polygons and back-facing polygons.
- Logical operators (GL_LOGIC_OP) cannot be selected for blending.
- GL_SRC_ALPHA_SATURATE can be selected for the destination in the glBlendFunc∗ blending functions, even though this is not allowed by the OpenGL ES specifications.

- Dithering is not applied.
- Logical operations are implemented to correspond with the OpenGL ES 1.1 specifications.

# 13.1. Alpha Test

The alpha test compares a fragment's alpha value with a reference value, to determine whether to pass the fragment to the next process or reject it.

The 3DS alpha test has features that correspond to the alpha test in OpenGL ES 1.1, but they are all controlled by reserved uniforms.

## 13.1.1. Reserved Uniform

The following reserved uniforms are used for the alpha test.

#### Enabling or Disabling the Alpha Test

To enable the alpha test, set the reserved uniform `dmp_FragOperation.enableAlphaTest` to `GL_TRUE`, using the `glUniform1i()` function. Disabled (set to `GL_FALSE`) by default.

#### Comparison Function

Use the `glUniform1i()` function to set the comparison function in the reserved uniform `dmp_FragOperation.alphaTestFunc`. You can specify one of the following eight values.

**Table 13-1. Comparison Functions for the Alpha Test**

| Setting Value | Comparison |
|---|---|
| `GL_NEVER` | Always rejected (never passes). |
| `GL_ALWAYS` (default) | Always passes. |
| `GL_LESS` | Passes if the alpha value is less than the reference value. |
| `GL_LEQUAL` | Passes if the alpha value is less than, or equal to, the reference value. |
| `GL_EQUAL` | Passes if the alpha value is equal to the reference value. |
| `GL_GEQUAL` | Passes if the alpha value is greater than, or equal to, the reference value. |
| `GL_GREATER` | Passes if the alpha value is greater than the reference value. |
| `GL_NOTEQUAL` | Passes if the alpha value is not equal to the reference value. |

#### Reference Value

Use the `glUniform1f()` function to set the reference value for the alpha test in the reserved uniform `dmp_FragOperation.alphaRefValue`. The reference value is clamped between 0.0 and 1.0 when it is used for comparisons.

The following table lists the reserved uniforms used for the alpha test.

**Table 13-2. Reserved Uniforms Used for the Alpha Test**

| Reserved Uniform | Type | Setting Value |
|---|---|---|
| `dmp_FragOperation.enableAlphaTest` | `bool` | Specifies whether to enable or disable the alpha test.<br>GL_TRUE<br>GL_FALSE (default) |
| `dmp_FragOperation.alphaRefValue` | `float` | Specifies the reference value to use with the alpha test.<br>0.0–1.0<br>0.0 by default. |
| `dmp_FragOperation.alphaTestFunc` | `int` | Specifies the comparison function for the alpha test.<br>GL_NEVER<br>GL_ALWAYS (default)<br>GL_LESS<br>GL_LEQUAL<br>GL_EQUAL<br>GL_GEQUAL<br>GL_GREATER<br>GL_NOTEQUAL |

# 13.2. Stencil Test

The stencil test compares a reference value with the value stored at a fragment's window coordinates in the stencil buffer to determine whether to pass the fragment to the next process or reject it.

The `glStencilFuncSeparate()` and `glStencilOpSeparate()` functions have not been implemented because the 3DS system does not distinguish between front-facing polygons and back-facing polygons.

## 13.2.1. How to Use

The stencil test is used in the same way it is used in OpenGL.

### Enabling or Disabling the Stencil Test

To enable or disable the stencil test, call the `glEnable()` or `glDisable()` function, passing `GL_STENCIL_TEST` for *cap*. To get the current setting, call the `glIsEnabled()` function, specifying `GL_STENCIL_TEST` for *cap*. The stencil test is disabled by default. When it is disabled, the stencil buffer is not changed and fragments are not rejected. If a stencil buffer has not been bound to the framebuffer, the stencil test is treated as if it were disabled. There is no performance difference between when both the stencil test and the depth test are enabled and when only one of the two is enabled.

### Comparison Function, Reference Value, and Mask

The stencil test has three elements: the comparison function, reference value, and mask, which are specified by the `glStencilFunc()` function.

**Code 13-1. Definition of the glStencilFunc Function**

```
void glStencilFunc(GLenum func, GLint ref, GLuint mask);
```

For the *ref* parameter, specify an integer to use as the reference value during comparisons. The reference value is handled as an unsigned number when it is compared by the stencil test, and it is clamped between 0 and 255 because the 3DS stencil buffer is always 8-bit. A value of 0 is used by default.

For the *mask* parameter, specify the mask. Bitwise AND operations apply the mask to both the reference value and the value in the stencil buffer to get the values to use during comparisons. A value of `0xFFFFFFFF` (all 1s) is used by default.

For the *func* parameter, specify the comparison function. You can specify one of the following eight values.

**Table 13-3. Comparison Functions for the Stencil Test**

| Setting Value | Comparison |
|---|---|
| GL_NEVER | Always rejected (never passes). |
| GL_ALWAYS (default) | Always passes. |
| GL_LESS | Passes if the reference value is less than the value in the stencil buffer. |
| GL_LEQUAL | Passes if the reference value is less than, or equal to, the value in the stencil buffer. |
| GL_EQUAL | Passes if the reference value is equal to the value in the stencil buffer. |
| GL_GEQUAL | Passes if the reference value is greater than, or equal to, the value in the stencil buffer. |
| GL_GREATER | Passes if the reference value is greater than the value in the stencil buffer. |
| GL_NOTEQUAL | Passes if the reference value is not equal to the value in the stencil buffer. |

## Handling Test Results

Using the `glStencilOp()` function, you can specify how the contents of the stencil buffer are modified as a result of both the stencil test (which compares the reference value with the stencil buffer) and the depth test, which will be described later.

**Code 13-2. Definition of the glStencilOp Function**

```
void glStencilOp(GLenum fail, GLenum zfail, GLenum zpass);
```

The *fail* parameter specifies how the contents of the stencil buffer are modified when a fragment is rejected by the stencil test. You can specify one of the following eight values.

**Table 13-4. Modifying Stencil Buffer Contents**

| Setting Value | Stencil Buffer Content |
|---|---|
| GL_KEEP (default) | Keep the current value. |
| GL_ZERO | Set the value equal to 0. |
| GL_REPLACE | Replace the value with the reference value. |
| GL_INCR | Increment the value by 1. This is never greater than 255. |
| GL_DECR | Decrement the value by 1. This is never less than 0. |
| GL_INVERT | Bitwise invert the value. |
| GL_INCR_WRAP | Increment the value by 1. This wraps to 0 after 255. |

| | |
|---|---|
| GL_DECR_WRAP | Decrement the value by 1. This wraps to 255 after 0. |

The *zfail* parameter configures how the contents of the stencil buffer are modified when a fragment is rejected by the depth test, and the *zpass* parameter configures how the contents of the stencil buffer are modified when a fragment passes the depth test. For these parameters, specify one of the eight values that is valid for the *fail* parameter, earlier.

# 13.3. Early Depth Test

The depth test compares a fragment's depth value with the value stored at the fragment's window coordinates in the depth buffer to determine whether to pass the fragment to the next process or reject it. As explained for the stencil buffer, the results of the depth test can also change the content of the stencil buffer.

## 13.3.1. How to Use

The depth test is used in the same way it is used in OpenGL.

#### Enabling or Disabling the Depth Test

To enable or disable the depth test, call the `glEnable` or `glDisable()` function, specifying `GL_DEPTH_TEST` for the *cap* parameter. To get the current setting, call the `glIsEnabled()` function, specifying `GL_DEPTH_TEST` for the *cap* parameter. The depth test is disabled by default. When it is disabled, the depth buffer is not modified, and fragments are not rejected. If a depth buffer has not been bound to the framebuffer, the depth test is treated as if it were disabled. There is no performance difference between when both the stencil test and the depth test are enabled and when only one of the two is enabled.

#### Comparison Function

Use the `glDepthFunc()` function to set the comparison function.

**Code 13-3. Definition of the glDepthFunc Function**

```
void glDepthFunc(GLenum func);
```

For *func*, specify one of the following eight values.

**Table 13-5. Comparison Functions for the Depth Test**

| Setting Value | Comparison |
|---|---|
| GL_NEVER | Always rejected (never passes). |
| GL_ALWAYS | Always passes. |
| GL_LESS (default) | Passes if the depth value is less than the value in the depth buffer. |
| GL_LEQUAL | Passes if the depth value is less than, or equal to, the value in the depth buffer. |
| GL_EQUAL | Passes if the depth value is equal to the value in the depth buffer. |

| | |
|---|---|
| GL_GEQUAL | Passes if the depth value is greater than, or equal to, the value in the depth buffer. |
| GL_GREATER | Passes if the depth value is greater than the value in the depth buffer. |
| GL_NOTEQUAL | Passes if the depth value is not equal to the value in the depth buffer. |

If a fragment is rejected as a result of the comparison, the depth buffer is not modified. If a fragment passes, its depth value overwrites the value in the depth buffer.

# 13.4. Blending

Blending combines a fragment's color (the source color) with the color stored at the fragment's window coordinates in the framebuffer (the destination color). The blended result is passed to the next process as the fragment's color.

## 13.4.1. How to Use

Blending is used in the same way it is used in OpenGL.

### Enabling or Disabling Blending

To enable or disable blending, call the `glEnable` or `glDisable()` function, specifying `GL_BLEND` for *`cap`*. To get the current setting, call the `glIsEnabled()` function, specifying `GL_BLEND` for *`cap`*. Blending is disabled by default. When it is disabled, blending is skipped.

Blending is considered to be disabled, either when a color buffer has not been bound to the framebuffer or when logical operations (described later) are enabled.

### Blend Equations

You can use the `glBlendEquation` or `glBlendEquationSeparate()` function to specify how a fragment's color (the source color) is combined with a framebuffer color (the destination color).

**Code 13-4. Definitions of the glBlendEquation* Functions**

```
void glBlendEquation(GLenum mode);
void glBlendEquationSeparate(GLenum modeRGB, GLenum modeAlpha);
```

The `glBlendEquation()` function allows you to specify equations for both the RGB and alpha components by using the *`mode`* parameter.

The `glBlendEquationSeparate()` function allows you to specify equations for the RGB and alpha components, separately, using the *`modeRGB`* and *`modeAlpha`* parameters, respectively. All of these parameters accept the following five values.

**Table 13-6. Blending Equations**

| Setting Value | RGB Components | Alpha Component |
|---|---|---|
| GL_FUNC_ADD (default) | R = Rs * Sr + Rd * Dr<br>G = Gs * Sg + Gd * Dg<br>B = Bs * Sb + Bd * Db | A = As * Sa + Ad * Da |

| | | |
|---|---|---|
| GL_FUNC_SUBTRACT | R = Rs * Sr + Rd * Dr<br>G = Gs * Sg + Gd * Dg<br>B = Bs * Sb + Bd * Db | A = As * Sa - Ad * Da |
| GL_FUNC_REVERSE_SUBTRACT | R = Rd * Sr - Rs * Dr<br>G = Gd * Sg - Gs * Dg<br>B = Bd * Sb - Bs * Db | A = Ad * Sa - As * Da |
| GL_MIN | R = min( Rs, Rd )<br>G = min( Gs, Gd )<br>B = min( Bs, Bd ) | A = min( As, Ad ) |
| GL_MAX | R = max( Rs, Rd )<br>G = max( Gs, Gd )<br>B = max( Bs, Bd ) | A = max( As, Ad ) |

Rs, Gs, Bs, and As represent the source color components.
Rd, Gd, Bd, and Ad represent the destination color components.
Sr, Sg, Sb, and Sa represent the source blend factors.
Dr, Dg, Db, and Da represent the destination blend factors.

GL_LOGIC_OP, which can be specified in OpenGL ES 2.0, is not supported.

### Source and Destination Blend Factors

You can use the glBlendFunc or glBlendFuncSeparate() function to specify the blend factors to apply to the source and destination.

**Code 13-5. Definitions of the glBlendFunc* Functions**

```
void glBlendFunc(GLenum sfactor, GLenum dfactor);
void glBlendFuncSeparate(GLenum srcRGB, GLenum dstRGB, GLenum srcAlpha,
                         GLenum dstAlpha);
```

The glBlendFunc() function allows you to specify both source and destination blend factors using *sfactor* and *dfactor*. The glBlendFuncSeparate() function allows you to specify the blend factors for the source RGB and alpha components using *srcRGB* and *srcAlpha*, and the blend factors for the destination RGB and alpha components using *dstRGB* and *dstAlpha*. All of these parameters accept the following 15 values.

**Table 13-7. Source and Destination Blend Factors**

| Setting Value | Color Blend Factor (Sr, Sg, Sb) or (Dr, Dg, Db) | Alpha Blend Factor (Sa or Da) |
|---|---|---|
| GL_ZERO (default for *dstRGB* and *dstAlpha*) | (0, 0, 0) | 0 |
| GL_ONE (default for *srcRGB* and *srcAlpha*) | (1, 1, 1) | 1 |
| GL_SRC_COLOR | (Rs, Gs, Bs) | As |
| GL_ONE_MINUS_SRC_COLOR | (1, 1, 1) - (Rs, Gs, Bs) | 1 - As |
| GL_DST_COLOR | (Rd, Gd, Bd) | Ad |
| GL_ONE_MINUS_DST_COLOR | (1, 1, 1) - (Rd, Gd, Bd) | 1 - Ad |
| GL_SRC_ALPHA | (As, As, As) | As |
| GL_ONE_MINUS_SRC_ALPHA | (1, 1, 1) - (As, As, As) | 1 - As |
| GL_DST_ALPHA | (Ad, Ad, Ad) | Ad |
| GL_ONE_MINUS_DST_ALPHA | (1, 1, 1) - (Ad, Ad, Ad) | 1 - Ad |
| GL_CONSTANT_COLOR | (Rc, Gc, Bc) | Ac |

| | | |
|---|---|---|
| GL_ONE_MINUS_CONSTANT_COLOR | (1, 1, 1) - (Rc, Gc, Bc) | 1 - Ac |
| GL_CONSTANT_ALPHA | (Ac, Ac, Ac) | Ac |
| GL_ONE_MINUS_CONSTANT_ALPHA | (1, 1, 1) - (Ac, Ac, Ac) | 1 - Ac |
| GL_SRC_ALPHA_SATURATE | (f, f, f) | 1 |

Rs, Gs, Bs, and As represent the source color.
Rd, Gd, Bd, and Ad represent the destination color.
Rc, Gc, Bc, and Ac represent the constant color.
f = min(As, 1 - Ad).

The 3DS system also allows you to specify GL_SRC_ALPHA_SATURATE for the destination.

#### Constant Color

You can specify the constant color using the glBlendColor() function.

**Code 13-6. Definition of glBlendColor Function**

```
void glBlendColor(GLclampf red, GLclampf green, GLclampf blue, GLclampf alpha);
```

Specify each of the RGB and alpha components using the *red*, *green*, *blue*, and *alpha* parameters. Specify a floating-point number between 0.0 and 1.0 for each component. By default, they are all 0.0 (0.0, 0.0, 0.0, 0.0).

## 13.5. Logical Operations

The final per-fragment operation is the logical operation, which is applied to fragment colors and framebuffer colors for an image. The result of the logical operation is written at the fragment's window coordinates in the framebuffer.

### 13.5.1. How to Use

Logical operations are used in the same way they are used in OpenGL ES 1.1.

#### Enabling or Disabling Logical Operations

To enable or disable logical operations, call the glEnable or glDisable() function, specifying GL_COLOR_LOGIC_OP for the *cap* parameter. To get the current setting, call the glIsEnabled() function, specifying GL_COLOR_LOGIC_OP for the *cap* parameter. Logical operations are disabled by default. When disabled, logical operations are skipped, but fragment colors are written to the framebuffer. Blending is disabled when logical operations are enabled.

#### Operation

Use the glLogicOp() function to specify the logical operation to be performed on the fragment color (the source) and the framebuffer color (the destination). To get the current setting, call the

`glGetIntegerv()` function, specifying `GL_LOGIC_OP_MODE` as the *pname* parameter. The default setting is `GL_COPY`.

**Code 13-7. Definition of the glLogicOp Function**

```
void glLogicOp(GLenum opcode);
```

Specify the logical operation for both the RGB and alpha components, by using the *opcode* parameter.

**Table 13-8. Logical Operations**

| Setting Value | Operation | C Notation |
|---|---|---|
| GL_CLEAR | 0 | 0 |
| GL_AND | s ∧ d | s & d |
| GL_AND_REVERSE | s ∧ ¬ d | s & ~d |
| GL_COPY (default) | s | s |
| GL_AND_INVERTED | ¬ s ∧ d | ~s & d |
| GL_NOOP | d | d |
| GL_XOR | s xor d | s ^ d |
| GL_OR | s ∨ d | s \| d |
| GL_NOR | ¬ (s ∨ d) | ~(s \| d) |
| GL_EQUIV | ¬ (s xor d) | ~(s ^ d) |
| GL_INVERT | ¬ d | ~d |
| GL_OR_REVERSE | s ∨ ¬ d | s \| ~d |
| GL_COPY_INVERTED | ¬ s | ~s |
| GL_OR_INVERTED | ¬ s ∨ d | ~s \| d |
| GL_NAND | ¬ (s ∧ d) | ~(s & d) |
| GL_SET | all 1 | pow(2, *n*) – 1 |

*n* is the number of bits in each component.

## 13.6. Masking the Framebuffer

You can apply a masking operation to the color (RGBA), stencil, and depth values that are written to the framebuffer by per-fragment operations. These values can be configured by the `glColorMask`, `glStencilMask`, and `glDepthMask()` functions, respectively.

**Code 13-8. Framebuffer Masking Functions**

```
void glColorMask(GLboolean red, GLboolean green, GLboolean blue,
                 GLboolean alpha);
void glStencilMask(GLuint mask);
void glDepthMask(GLboolean flag);
```

You can specify whether to allow (`GL_TRUE`) or deny (`GL_FALSE`) the writing of each RGBA color value and depth value. These values are all set to `GL_TRUE` by default.

You can modify the stencil masking value. This masking value is independent of the masking value for the stencil test. It is `0xFFFFFFFF` (all 1s) by default.



# 14. Framebuffer Operations

Framebuffer operations read, copy, and otherwise manipulate framebuffer data.

The features supported by the 3DS system are restricted as follows.

- The `glDrawBuffer()` function has not been implemented.
- The `glReadBuffer()` function has not been implemented.
- The `glDrawPixels()` function has not been implemented.
- The `glReadPixels()` function can get the contents of the color, depth, and stencil buffers.
- The `glCopyPixels()` function has not been implemented.
- The `glPixelStore*()` functions have not been implemented.

## 14.1. Reading Pixels

You can use the `glReadPixels()` function to take the contents of a render buffer (the color, depth, or stencil buffer) bound to the framebuffer and place it into memory. You can also read the contents of the depth and stencil buffers into memory, using a combined format.

**Code 14-1. glReadPixels Function**

```
void glReadPixels(GLint x, GLint y, GLsizei width, GLsizei height,
                  GLenum format, GLenum type, void* pixels);
```

For *x* and *y*, specify the starting coordinates (the lower-left corner) of the rectangular region to capture. For *width* and *height*, specify the width and height of the rectangular region. A `GL_INVALID_VALUE` error is generated if any of these values are negative. A `GL_INVALID_OPERATION` error is generated when the sum of *x* and *width* exceeds the render buffer width, or when the sum of *y* and *height* exceeds the render buffer height.

Using a combination of *format* and *type*, you can specify the format of the image to capture.

**Table 14-1. Formats That Can Be Specified for Reading**

| format | type | Format | Bits |
|---|---|---|---|
| `GL_RGBA` | `GL_UNSIGNED_BYTE` | RGBA8 | 32 |
| | `GL_UNSIGNED_SHORT_4_4_4_4` | RGBA4 | 16 |
| | `GL_UNSIGNED_SHORT_5_5_5_1` | RGBA5551 | 16 |
| `GL_RGB` | `GL_UNSIGNED_BYTE` | RGB8 | 24 |
| | `GL_UNSIGNED_SHORT_5_6_5` | RGB565 | 16 |

| | | | |
|---|---|---|---|
| GL_BGRA | GL_UNSIGNED_BYTE | BGRA8 | 32 |
| | GL_UNSIGNED_SHORT_4_4_4_4 | BGRA4 | 16 |
| | GL_UNSIGNED_SHORT_5_5_5_1 | BGRA5551 | 16 |
| GL_LUMINANCE_ALPHA | GL_UNSIGNED_BYTE | LA8 | 16 |
| GL_LUMINANCE | GL_UNSIGNED_BYTE | L8 | 8 |
| GL_ALPHA | GL_UNSIGNED_BYTE | A8 | 8 |
| GL_DEPTH_COMPONENT | GL_UNSIGNED_INT | Depth | 32 |
| | GL_UNSIGNED_INT24_DMP | Depth | 24 |
| | GL_UNSIGNED_SHORT | Depth | 16 |
| | GL_UNSIGNED_BYTE | Depth | 8 |
| GL_STENCIL_INDEX | GL_UNSIGNED_BYTE | Stencil | 8 |
| GL_DEPTH24_STENCIL8_EXT | GL_UNSIGNED_INT | Depth + Stencil | 32 |

A GL_INVALID_ENUM error is generated if you specify a combination that is not in the table.

The captured image is stored in *pixels*, using the specified format.

Images taken from the color buffer are stored in linear format, using a pixel byte order that is OpenGL-compliant.

Images read from the depth buffer are also stored in linear format, but the pixel data is ordered starting with the least-significant bits. For example, an image captured using a 24-bit format is stored in memory one byte at a time, starting with the least-significant byte and ending with the most-significant byte.

Images read from the stencil buffer are stored in linear format, with one byte per pixel.

Images read from the depth and stencil buffers in a combined format are also stored in linear format, but the pixel data is ordered starting with the least-significant bits. 24 bits of depth data are stored first, one byte at a time, from the least-significant to the most-significant byte, followed by 8 bits of stencil data.

A GL_INVALID_FRAMEBUFFER_OPERATION error is generated if an error related to the framebuffer occurs (for example, if a color buffer has not been bound to the framebuffer). A GL_OUT_OF_MEMORY error is generated when temporary memory fails to be allocated within the library.

## 14.2. Copying Pixels

The glCopyPixels() function does not support copying data to other framebuffers.

You can call the glCopyTexImage2D() and glCopyTexSubImage2D() functions to copy the content of the color buffer to a texture. For more information, see 7.5.1. Copying From the Color Buffer.

## 14.3. Texture Rendering

You can bind a texture to the framebuffer as a render buffer. For more information, see 7.6. Specifying a Texture as the Render Target.

# 14.4. Clearing the Framebuffer

In OpenGL, the scissor test and masking affect how the `glClear()` function clears (every buffer bound to) the framebuffer. On a 3DS system, however, the scissor test and masking are not included in the graphics pipeline and have no effect.

You can bind three render buffers to the framebuffer, as explained in 3.3.1. Render Buffer Allocation: a color buffer (which could be a texture or a buffer used to render gas), a depth buffer, and a stencil buffer. Index colors are not supported.

To clear a buffer, call the `glClear()` function with the buffer's corresponding bit specified as an argument.

**Code 14-2. Definition of the glClear Function**

```
void glClear(GLbitfield mask);
```

You can specify the following values as a bit field in *`mask`*. To clear multiple buffers simultaneously, specify the bit field using bitwise OR operations.

**Table 14-2. Specifying the Buffers to Clear**

| mask | Buffer to Clear |
|---|---|
| `GL_COLOR_BUFFER_BIT` | Color buffer. |
| `GL_DEPTH_BUFFER_BIT` | Depth buffer. |
| `GL_STENCIL_BUFFER_BIT` | Stencil buffer. This must be specified together with `GL_DEPTH_BUFFER_BIT`. |

You cannot clear the 3DS stencil buffer alone because it is combined with the depth buffer (`GL_DEPTH24_STENCIL8_EXT`).

## 14.4.1. Clear Color

You can use the `glClearColor()` function to specify the color to use when clearing the color buffer.

**Code 14-3. Specifying the Clear Color**

```
void glClearColor(GLclampf red, GLclampf green, GLclampf blue, GLclampf alpha);
```

Each of the specified color components is clamped between 0.0 and 1.0. By default, all components are set to 0.0.

## 14.4.2. Clear Depth

You can use the `glClearDepthf()` function to specify the depth value to use when clearing the depth buffer.

**Code 14-4. Specifying the Clear Depth**

```
void glClearDepthf(GLclampf depth);
```

The specified values are clamped between 0.0 and 1.0. The default value is 1.0.

### 14.4.3. Clear Stencil

You can use the `glClearStencil()` function to specify the value to use when clearing the stencil buffer.

**Code 14-5. Specifying the Clear Stencil**

```
void glClearStencil(GLint s);
```

Specify a value between `0` and 255 to use for clear operations. The default value is `0`.



# 15. Other

## 15.1. glFinish and glFlush Functions

The `glFinish()` function is the same as the `glFlush()` function on a 3DS system.

**Code 15-1. Definitions of the glFinish and glFlush Functions**

```
void glFinish(void);
void glFlush(void);
```

## 15.2. Differences From OpenGL ES

Most features and functions follow the OpenGL ES specifications, but there are some differences. For example, some OpenGL ES functions and features have not been implemented, and others have restrictions.

**Table 15-1. Differences by Feature**

| Features | Differences |
|---|---|
| `GL_ALPHA_TEST` | Reserved uniforms are used to control the alpha test. |
| `GL_CLIP_PLANEi` | Reserved uniforms are used to control clipping. |

| | |
|---|---|
| GL_COLOR_LOGIC_OP | Logical operations are not present in OpenGL ES 2.0, but the equivalent features from OpenGL ES 1.1 have been implemented for 3DS. |
| GL_DITHER | Dithering is not supported. |
| GL_FOG | Reserved uniforms are used to control fog. Fog coefficients are set by a lookup table that takes depth values in window coordinates as input, rather than by the distance from the viewpoint. |
| GL_INDEX_LOGIC_OP | Index colors are not supported. |
| GL_LIGHTING | Reserved uniforms are used to control lighting. 3DS lighting is a per-fragment operation. |
| GL_POLYGON_OFFSET_FILL<br>GL_POLYGON_OFFSET_LINE<br>GL_POLYGON_OFFSET_POINT | Only GL_POLYGON_OFFSET_FILL is supported for controlling polygon offsets. |
| GL_SAMPLE_COVERAGE<br>GL_SAMPLE_ALPHA_TO_COVERAGE | Multisampling is not supported. |
| GL_SCISSOR_TEST | The scissor test is run as a subprocess of rasterization. |
| GL_TEXTURE_2D<br>GL_TEXTURE_CUBE_MAP | Reserved uniforms are used to control texture units. |

**Table 15-2. Differences by Function**

| Function | Differences |
|---|---|
| Shader Functions | |
| glBufferData | Only GL_STATIC_DRAW can be specified for usage. |
| glCompileShader | This function is not implemented. |
| glCreateProgram | This function uses a (13-bit) namespace that is independent of shader objects. |
| glCreateShader | This function uses a namespace that is independent of program objects. |
| glDrawArrays(), glDrawElements() | GL_POINTS, GL_LINES, GL_LINE_STRIP, and GL_LINE_LOOP are not supported. You must use the geometry shader to render points and lines. |
| glGetProgramInfoLog | This function is not implemented. |
| glGetShaderInfoLog | This function is not implemented. |
| glGetShaderSource() | This function is not implemented. |
| glGetShaderPrecisionFormat | This function is not implemented. |
| glLineWidth | This function is not implemented. |
| glReleaseShaderCompiler | This function is not implemented. |
| glShaderBinary | Only GL_PLATFORM_BINARY_DMP can be specified for *binaryformat*. |
| glShaderSource | This function is not implemented. |
| glValidateProgram | This function does nothing when it is called. |
| glVertexAttribPointer | Neither GL_FIXED nor GL_UNSIGNED_SHORT can be specified for *type*. The data pointer must be 4-byte aligned when specifying GL_FLOAT for *type*, or 2-byte aligned when specifying GL_SHORT. This function does not support normalization specified via arguments. |
| Viewport Functions | |
| glViewport | You cannot specify a negative value for *x* or *y*. |
| Texture Functions | |
| glActiveTexture | An error is generated when GL_TEXTURE3 is specified. |
| glCompressedTexImage2D | The width and height must be specified as powers of 2 (from 16 |

| | |
|---|---|
| | through 1024). |
| `glCompressedTexSubImage2D` | This function is not implemented. |
| `glGenerateMipmap` | This function is not implemented. |
| `glTexImage1D`<br>`glTexSubImage1D` | These functions are used to load lookup tables. One-dimensional textures are not supported. |
| `glTexImage2D` | The width and height must be specified as powers of 2 (from 8 through 1024). |
| `glTexSubImage2D` | This function is not implemented. |
| Render Buffer Functions | |
| `glClear` | Index colors are not supported. The stencil buffer must be cleared at the same time as the depth buffer. This function is unaffected by the scissor test and masking. |
| `glCopyPixels` | This function is not implemented. |
| `glDrawBuffer` | This function is not implemented. |
| `glDrawPixels` | This function is not implemented. |
| `glFramebufferRenderbuffer` | Because the stencil buffer is combined with the depth buffer, `GL_DEPTH_STENCIL_ATTACHMENT` must be specified as the attachment point for the stencil buffer. |
| `glFramebufferTexture2D` | The color buffer is the only supported attachment point. |
| `glPixelStorei` | This function is not implemented. |
| `glReadBuffer` | This function is not implemented. |
| `glRenderbufferStorage` | There are restrictions on which formats can be specified. The only format that can be used for the stencil buffer is `GL_DEPTH24_STENCIL8_EXT`. |
| `glSampleCoverage` | This function is not implemented. |
| `glStencilFuncSeparate`<br>`glStencilOpSeparate`<br>`glStencilMaskSeparate` | This function is not implemented. |
| Blending Functions | |
| `glBlendEquation`<br>`glBlendEquationSeparate` | `GL_LOGIC_OP`, which can be specified in OpenGL ES 2.0, is not supported. |
| `glBlendFunc`<br>`glBlendFuncSeparate` | `GL_SRC_ALPHA_SATURATE` can also be specified for the destination. |
| Other | |
| `glFinish` | This function does the same thing as `glFlush`. |
| `glFlush` | There is no difference until `glFinish` is called. |
| `glHint` | This function is not implemented. |
| `glPolygonOffset` | The value set for *`factor`* is ignored. |

## 15.3. Improving Buffer Access Performance

When you are not using either the color, depth, or stencil buffer, you can explicitly disable that buffer's features to reduce unnecessary processing. Because the same buffer is used for the 3DS depth and stencil buffers, however, settings to access either one have the same performance as settings to access both.

The following sections describe the conditions for accessing each buffer. Avoid these conditions to prevent unnecessary accesses from occurring.

### 15.3.1. Write Access to the Color Buffer

These accesses occur when the `glColorMask()` function specifies a value of `GL_TRUE` for any component.

### 15.3.2. Read Access to the Color Buffer

These accesses occur when write accesses to the color buffer occur and any of the following conditions are met.

- `GL_BLEND` has been enabled by the `glEnable()` function.
- A call of the `glColorMask()` function has not specified the same values for all components.
- `GL_COLOR_LOGIC_OP` has been enabled by the `glEnable()` function.

### 15.3.3. Write Access to the Depth Buffer

These accesses occur when `GL_DEPTH_TEST` has been enabled by the `glEnable()` function and `GL_TRUE` has been specified by the `glDepthMask()` function.

### 15.3.4. Read Access to the Depth Buffer

These accesses occur when `GL_DEPTH_TEST` has been enabled by the `glEnable()` function.

### 15.3.5. Write Access to the Stencil Buffer

These accesses occur when `GL_STENCIL_TEST` has been enabled by the `glEnable()` function, and a nonzero masking setting is used in the `glStencilMask()` function.

### 15.3.6. Read Access to the Stencil Buffer

These accesses occur when `GL_STENCIL_TEST` has been enabled by the `glEnable()` function.

## 15.4. Improving CPU Performance

You may be able to improve processing speed by paying attention to the following points when you implement your application.

- Link as many vertex shader objects together as possible. You can link multiple shader objects together to generate a single shader binary. It is less resource intensive to switch between shader objects when they are linked to the same shader binary than when they are linked to different shader binaries.
- Ensure that applications keep the uniform location values obtained by the `glGetUniformLocation()` function, and use them repeatedly. Location values are static after the `glLinkProgram()` function is called. They do not change until `glLinkProgram` is called again.
- Do not call the `nngxSplitDrawCmdlist()` function unnecessarily because it generates a split command each time it is called. For example, when the `nngxTransferRenderImage()` function is called, it generates a split command internally. Calling `nngxSplitDrawCmdlist` immediately afterwards would generate an unnecessary command.
- Use a vertex buffer, whenever possible, to send vertex attribute data to a shader. If you do not use a vertex buffer, the CPU accumulates vertex data in the 3D command buffer, and CPU processing increases significantly.
- Use a texture collection or a vertex state collection when the same texture or vertex buffer is used repeatedly in a particular rendering pass. By binding all textures and setting all vertex arrays at the same time, you can reduce the cost of function calls.
- When executing the same shader object with different uniform settings, it is less resource intensive to attach that shader object to multiple program objects (each with its own uniform settings) and then switch between the program objects than it is to switch between many uniform settings on a single program object. The reason is that uniform values are saved for each program object.
- Do not delete and regenerate lookup table objects frequently. After the lookup table data is loaded by the `glTexImage1D()` function, it is converted into an internal hardware format each time the lookup table object is used.

## 15.5. Using Vertex Buffers

When a vertex buffer is used, the GPU geometry pipeline loads vertex data. When a vertex buffer is not used, however, the CPU sorts the vertex arrays according to the vertex index arrays, converts all vertex data into 24-bit floating-point values, and then fills the command buffer. This process imposes a considerably high processing load on the CPU, and also decreases the efficiency with which the GPU geometry pipeline loads data. This also requires a larger command buffer. All vertex data is converted into 24-bit floating-point numbers for the x, y, z, and w components when it is loaded into the command buffer, which must be able to hold at least 12 bytes multiplied by both the number of vertices and the number of vertex attributes.

Vertex buffers can be processed more quickly when they are placed in VRAM than when they are placed in main (device) memory. Splitting them between VRAM and main memory results in the same speed as placing them in main memory.

### 15.5.1. Data Structure for Vertex Arrays

A vertex array can either be structured as an interleaved array, which is an array of structures that contain multiple vertex attributes, or as an independent array, which is an array of single vertex

attributes.

When you use a vertex buffer, interleaved arrays are more efficient at loading vertex data than independent arrays. The time spent loading vertex data is often hidden by the time spent for processing later on, such as in the vertex shader and during rasterization. When the vertex buffer is placed in main memory, however, by making the loading of data more efficient, you can sometimes reduce the cost of data access and speed up processing.

## 15.6. Getting the Starting Address for Each Buffer Type

You can get the starting addresses of the data regions allocated for each texture object, vertex buffer object, and render buffer object.

The GPU can directly access all obtained addresses. You cannot get the address of the copy region that is generated by the driver.

**Code 15-2. Getting the Starting Address for Each Buffer Type**

```
void glGetTexParameteriv(GLenum target, GLenum pname, GLint* params);
void glGetBufferParameteriv(GLenum target, GLenum pname, GLint* params);
void glGetRenderbufferParameteriv(GLenum target, GLenum pname, GLint* params);
```

To get the texture address, call the `glGetTexParameteriv()` function and specify `GL_TEXTURE_DATA_ADDR_DMP` for *`pname`*.

To get the vertex buffer address, call the `glGetBufferParameteriv()` function and specify `GL_BUFFER_DATA_ADDR_DMP` for *`pname`*.

To get the render buffer address, call the `glGetRenderbufferParameteriv()` function and specify `GL_RENDERBUFFER_DATA_ADDR_DMP` for *`pname`*.

These functions allow you to get various types of information, depending on the value passed in for *`pname`*. For more information about the information that can be obtained, see the *CTR-SDK API Reference*.

## 15.7. Number of Bytes Loaded for Various Data Types

This section describes the number of bytes that the GPU loads from vertex buffers, textures, and command buffers in a single operation.

### 15.7.1. Vertex Buffers

The number of bytes loaded concurrently from a vertex buffer depends on the order of the vertex indices.

The `glDrawElements()` function loads 16 vertex indices concurrently from an index array, sorts them, and then loads data from the vertex array in the same order as the sorted indices. Consecutive data is loaded from the vertex array for any consecutive vertex indices.

Multiple vertex attributes are loaded as an interleaved array from a vertex array if: (1) the array

contains vertex attributes and has been enabled by the `glEnableVertexAttribArray()` function, and (2) the driver interprets the array as an interleaved array (a vertex array that combines multiple vertex attributes) based on the information specified by the `glVertexAttribPointer()` function.

Burst reads of up to 256 bytes are used to load consecutive data from a vertex array. If there are more than 256 bytes to be loaded, they are read 256 bytes at a time. Data is read at least 16 bytes at a time, even for non-consecutive data.

The `glDrawArrays()` function processes data just like an index array, using consecutive numbers starting at 0.

### 15.7.2. Textures

The number of bytes that are loaded at one time depends on the texture format. The following table shows the number of bytes that are transferred from VRAM, avoiding the texture cache.

**Table 15-3. Texture Format and Loading Units**

| format | type | Bytes |
|---|---|---|
| GL_RGBA<br>GL_RGBA_NATIVE_DMP | GL_UNSIGNED_BYTE | 128 |
| | GL_UNSIGNED_SHORT_5_5_5_1 | 64 |
| | GL_UNSIGNED_SHORT_4_4_4_4 | 64 |
| GL_RGB<br>GL_RGB_NATIVE_DMP | GL_UNSIGNED_BYTE | 96 |
| | GL_UNSIGNED_SHORT_5_6_5 | 64 |
| GL_LUMINANCE_ALPHA<br>GL_LUMINANCE_ALPHA_NATIVE_DMP | GL_UNSIGNED_BYTE | 64 |
| | GL_UNSIGNED_BYTE_4_4_DMP | 32 |
| GL_LUMINANCE<br>GL_LUMINANCE_NATIVE_DMP | GL_UNSIGNED_BYTE | 32 |
| | GL_UNSIGNED_4BITS_DMP | 16 |
| GL_ALPHA<br>GL_ALPHA_NATIVE_DMP | GL_UNSIGNED_BYTE | 32 |
| | GL_UNSIGNED_4BITS_DMP | 16 |
| GL_HILO8_DMP<br>GL_HILO8_DMP_NATIVE_DMP | GL_UNSIGNED_BYTE | 64 |
| GL_ETC1_RGB8_NATIVE_DMP | - | 128 |
| GL_ETC1_ALPHA_RGB8_A4_NATIVE_DMP | - | 32 |

### 15.7.3. Command Buffer

A command buffer loads 128 bytes at a time.

## 15.8. Block-Shaped Noise Is Rendered on Some Pixels

Data in the 3DS framebuffer is processed 4×4 pixels at a time. These blocks of pixels are called *block addresses* and are also used to manage the framebuffer cache. Tag information in the cache is cleared at several times, including when the `glFinish`, `glFlush`, or `glClear()` function is called;

when the framebuffer-related GPU state (NN_GX_STATE_FRAMEBUFFER, NN_GX_STATE_FBACCESS) is validated; and when the command list is split by the nngxSplitDrawCmdlist() function. Cache tags are initialized with their default value of 0x3FFF after tag information in the cache is cleared. Consequently, any pixels that you attempt to render immediately afterward at the same block address as the default cache tag value (0x3FFF) mistakenly hit the cache. As a result, an incorrect color is applied to the pixels.

Block addresses are assigned consecutively, beginning at 0, in 16-pixel blocks from the starting address of the framebuffer (the color buffer, depth buffer, and stencil buffer). Because addresses are assigned to data that has been laid out in the GPU render format, pixel locations in a rendered image correspond to different block addresses in block 8 mode and in block 32 mode.

The problem described in this section is triggered by pixels that are assigned a block address of 0x3FFF. The problem does not occur when the total number of framebuffer blocks is less than or equal to 0x3FFF (in other words, when the total number of framebuffer pixels is less than or equal to 0x3FFF×16, or 262,128 pixels). This is equivalent to a 512×512 rectangle). This problem also does not occur when there are no read accesses on the color buffer, depth buffer, or stencil buffer.

**Note:** Cache tag information is also cleared when a value of 1 is written to GPU register 0x0110.

For more information about accessing GPU registers directly and controlling the GPU state, block mode, and framebuffer read access, see the *3DS Programming Manual: Advanced Graphics*.

## 15.8.1. Relationship Between Pixels and Block Addresses

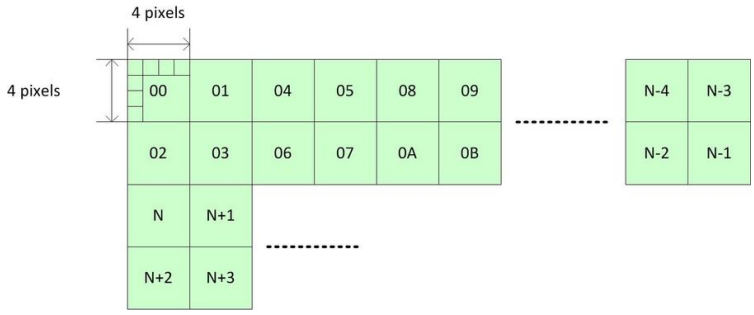
As mentioned previously, block addresses begin at 0 and are assigned in ascending order, 16 pixels at a time, from the starting addresses of the color buffer and depth/stencil buffer, which are laid out in the GPU render format. Unlike the origin for the glViewport() function, the buffer addresses start with the pixels at the upper-left corner of the image to render. Note, too, that the image width (the horizontal direction) corresponds to the shorter edge of the LCD.

Because there are different ways to assign addresses, the block mode changes which block address in the cache corresponds to the pixels on an image.

### 15.8.1.1. Block 8 Mode

Block address 0 corresponds to the 4×4 block of pixels at the upper-left corner of the rendered image; block address 1 corresponds to the 4×4 block of pixels immediately to the right of block address 0; block address 2 corresponds to the 4×4 block of pixels immediately below block addresses 0; and block address 3 corresponds to the 4×4 block of pixels immediately below block address 1. Block addresses increase to the right 8×8 pixels at a time. After they reach the edge of the image, they continue from the left edge of the image on the next row.

**Figure 15-1. Block Address Assignments in Block 8 Mode**



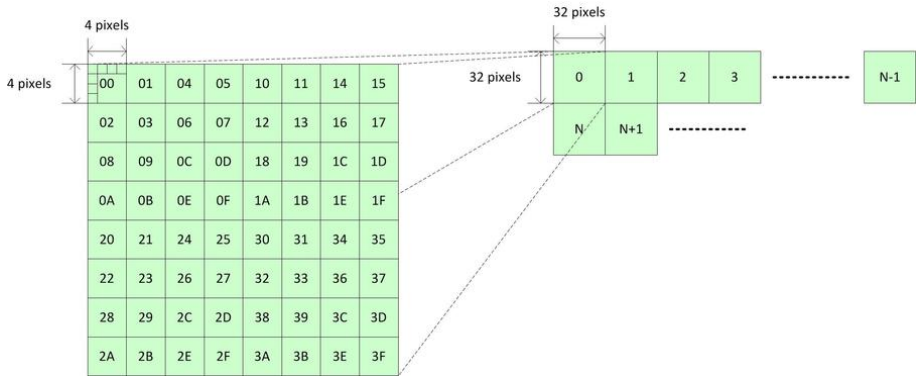

The value of *N* in the figure is calculated by taking one-quarter the width of the framebuffer (in pixels) and multiplying it by two.

### 15.8.1.2. Block 32 Mode

In block 32 mode, addresses are assigned in metablocks of 32×32 pixels. Metablock address 0 corresponds to the 32×32-pixel region at the upper-left corner of the rendered image, and metablock address 1 corresponds to the 32×32-pixel region immediately to its right. Metablock addresses increase to the right 32×32 pixels at a time. After they reach the edge of the image, they continue from the left edge of the image on the next row.

As the following figure shows, the block addresses of the pixels are arranged in a zigzag pattern within each metablock, starting with the 4×4-pixel block at the upper-left corner. To find the block address of a single pixel in the image, multiply its metablock address by `0x40`, and then add its block address within the metablock.

**Figure 15-2. Block Address Assignments in Block 32 Mode**




The left side of the figure shows the block addresses (in hexadecimal) for pixels within a metablock. The right side of the figure shows the metablock addresses for the entire image.

## 15.8.2. Workaround #1

This problem does not occur when the framebuffer has no more than 262,128 pixels (the product of its width and height). In other words, you can work around this problem by using a framebuffer that is no larger than necessary.

Note that the problem does not occur with a framebuffer that has the same size as one of the LCDs —240×400 (96,000 pixels) or 240×320 (76,800 pixels)—because the total number of pixels does not exceed 262,128.

Although you specify the framebuffer size with the `glRenderbufferStorage()` function, if you

allocate a large framebuffer and only use a part (240×400 region) of it, you can avoid this problem by using no more than the minimum necessary size for the allocated framebuffer region.

## 15.8.3. Workaround #2

You can work around this problem by adjusting the size of the framebuffer so that the problematic pixels at block address 0x3FFF are located outside of the rendering region.

For example, when you allocate a 480×800 framebuffer (as you would to apply 2×2 antialiasing in block 8 mode), block address `0x3FFF` is assigned to the 44-pixel region whose upper-left corner is located at pixel coordinates of (124, 548). If you were to extend the size of the framebuffer by 32 pixels to 512×800, however, block address `0x3FFF` would be assigned to the 4×4-pixel region whose upper-left corner is located at pixel coordinates (508, 508). By configuring the viewport to display only the 480×800 region on the left side of the framebuffer, you can avoid these problematic pixels.

One disadvantage of this method is that it requires you to allocate a framebuffer that is larger than necessary, which wastes VRAM. However, it is a simple workaround that only involves adjusting the framebuffer size.

For more information about how differences in the block mode and framebuffer size affect the pixel coordinates to which block address `0x3FFF` is assigned, see 15.8.1. Relationship Between Pixels and Block Addresses.

## 15.8.4. Workaround #3

You can work around this problem by rendering several pixels that are not at block address `0x3FFF` to change the content of cache tags immediately after they have been cleared.

To change the content of the cache tags, you must render four pixels at specific block addresses. When both the color buffer and depth/stencil buffer have been configured to be read, these four pixels must each have a different block address, for which the lower three bits are all 1 (`0x7`). When only one buffer (either the color buffer or the depth/stencil buffer) has been configured to be read, these four pixels must each have a different block address, for which the lower four bits are all 1 (`0xF`).

For example, assume that pixels are rendered at the following block addresses immediately after cache tags are cleared: `0x00`, `0x01`, `0x0F`, `0x02`, `0x1F`, `0x03`, `0x0F`, `0x2F`, and `0x3F`.

> Block addresses `0x00` and `0x01` do not count because their lower four bits are not `0xF`.

Block address `0x0F` is only counted once, even though pixels are rendered there twice. In this example, the workaround is only effective after pixels have been rendered at block addresses `0x0F`, `0x1F`, `0x2F`, and `0x3F`. If pixels at block address `0x3FFF` are rendered before the pixels at block address `0x3F`, the problem would occur.

You can work around this problem by rendering dummy polygons, with pixels that meet these conditions, immediately after cache tags are cleared, given the following caveats. The following are valid dummy pixels.

- Pixels that fail the depth test, stencil test, or alpha test. If you use settings that cause dummy pixels to always fail these tests (for example, by specifying `GL_NEVER` for the depth test function), make sure that you restore the original depth test function when you resume normal rendering. Note that the cache flush command (a command that writes to register `0x111`) would be required at this time.
- Pixels that do not affect the color buffer when they are rendered because of alpha blend

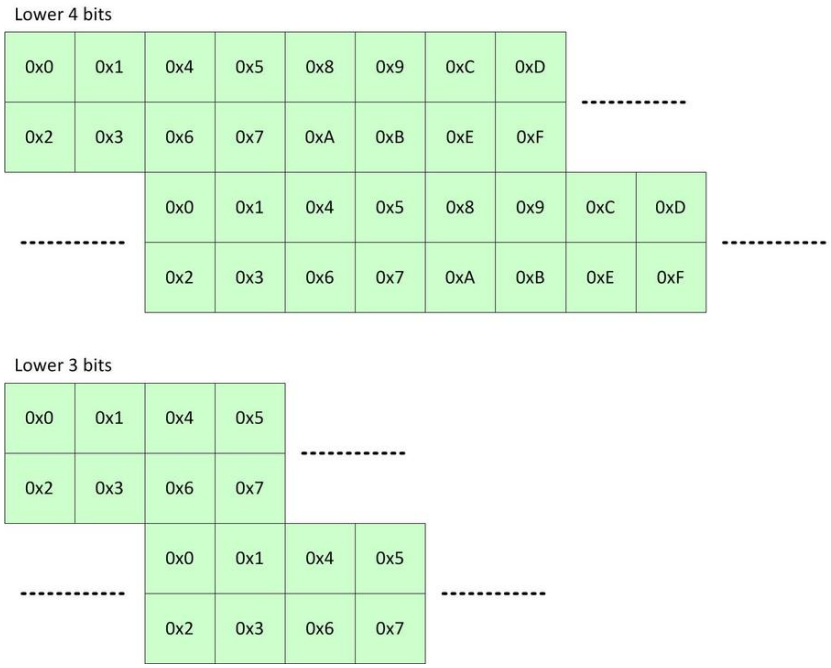
settings.

- The following are not valid dummy pixels. Pixels that are clipped by the view volume or user-defined clipping planes.
- Pixels that are dropped by the scissor test.
- Pixels that are dropped by the early depth test.

## 15.8.4.1. Block 8 Mode

When you render a dummy polygon to work around this problem, you must choose pixels at block addresses whose lower four bits (or lower three bits) are all 1. If you look at how block addresses are arranged in block 8 mode, the lower four bits of the block address follow the same 32×8-pixel pattern repeated horizontally, and the lower three bits of the block address follow the same 16×8-pixel pattern repeated horizontally. However, depending on the framebuffer width, these patterns may be shifted horizontally by eight pixels, for every eight pixels vertically.

**Figure 15-3. Block Address Ordering in Block 8 Mode**




The following table shows rectangle sizes that meet the conditions for a dummy polygon. Note that the rectangle with the smallest possible area must be placed so that the pixels at its four corners have block addresses that meet the necessary conditions.

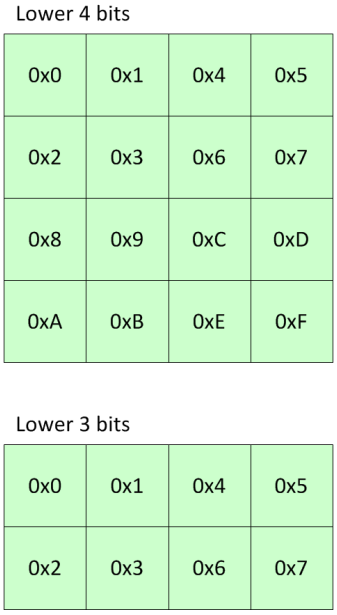
**Table 15-4. Rectangles to Render as Dummy Polygons in Block 8 Mode**

| Rectangle Shape/Conditions | Lower 4 Bits Are All 1 | Lower 3 Bits Are All 1 |
|---|---|---|
| Rectangle with the smallest possible area (cannot be placed anywhere). | 94×1 | 46×1 |
| Rectangle that can be placed anywhere. | 125×5<br>29×29 | 61×5<br>13×29 |

## 15.8.4.2. Block 32 Mode

When you render a dummy polygon to work around this problem, you must choose pixels at block addresses whose lower four bits (or lower three bits) are all 1. If you look at how block addresses are arranged in block 32 mode, the lower four bits of the block address follow the same 32×32-pixel pattern repeated horizontally, and the lower three bits of the block address follow the same 32×16-pixel pattern repeated horizontally.

**Figure 15-4. Block Address Ordering in Block 32 Mode**




The following table shows rectangle sizes that meet the conditions for a dummy polygon. Note that the rectangle with the smallest possible area must be placed so that the pixels at its four corners have block addresses that meet the necessary conditions.

**Table 15-5. Rectangles to Render as Dummy Polygons in Block 32 Mode**

| Rectangle Shape/Conditions | Lower 4 Bits Are All 1 | Lower 3 Bits Are All 1 |
|---|---|---|
| Rectangle with the smallest possible area (cannot be placed anywhere). | 46×1<br>14×14 | 46×1<br>14×6 |
| Rectangle that can be placed anywhere. | 61×13<br>29×29 | 61×5<br>29×13 |

## 15.9. Lines Are Rendered in Error and the Region Following the Framebuffer Is Corrupted

When rendering an extremely small polygon that has a right edge close to the window's x-coordinate of 0, the system sometimes renders lines in error. This phenomenon is caused by coordinate wraparound when the pixel's x-coordinate becomes negative, due to calculation errors on polygon pixel generation. Because of the extremely large x-coordinate value, the system renders a polygon with extremely elongated dimensions in the positive x direction.

This phenomenon can also cause corruption in the memory region outside of the rendering memory region. The wrapped x-coordinate becomes 1023, and the system generates pixels from (0, y) through (1023, y), regardless of the size of the rendering region. In other words, the system

generates pixels with x-coordinates from 0 through 1023, even when the rendering region size is set to a width smaller than 1024, such as 256×256. The framebuffer is accessed at the addresses calculated from the pixel's (x, y) coordinates and the width of the rendering region, but the raw x-coordinate is used for address calculation, even when it is outside of the rendering region's width. Consequently, depending on the y-coordinate value, the system might write pixel color data to memory addresses following the last address of the rendering region.

> **Note:** Memory corruption does not happen when the rendering region is cropped by using a scissor test. We recommend configuring a scissor test to avoid possible memory corruption. Conducting a scissor test does not entail any penalty in terms of GPU performance.

The conditions under which this phenomenon occurs depend solely on the window coordinates. For any set of window coordinates, the problem either always occurs or never occurs. This problem occurs in relation to the view volume or polygons generated as a result of clipping, so it occurs even when the original polygon itself is large, provided that the polygon protrudes beyond the edge of the screen when the window's x-coordinate is 0, producing an extremely small area contained within the view volume.

One workaround for this issue is to adjust the x-coordinate in the vertex shader. The clipping x-coordinate calculated by the vertex shader is clipped from –w to w, so x values close to –w indicate a vertex close to the screen edge at the window's x-coordinate of 0. You can avoid the erroneous lines by moving any such vertices—vertices located close to the edge of the screen where the x-coordinate is 0—away from the edge of the screen by adjusting the x value in the -w direction. This workaround only changes the vertex coordinate by no more than one pixel, so it has almost no effect on rendering results.

When processing this in the vertex shader, handle the x value after applying a projection transform (the value written to the output register as the vertex coordinate x value) as follows.

**Code 15-3. Processing to Avoid Rendering Lines in Error**

```
if ( -w < x && x < -w * (1-epsilon) ) x = -w;
```

These x and w values are the x and w values of the vertex coordinate after the projection transformation. The `epsilon` value is a variable for adjustment that is to be specified as appropriate for the scene you are rendering.

The following code section is a sample implementation of a vertex shader. Instructions starting from `mul` are included to avoid displaying lines in error.

**Code 15-4. Workaround Implementation Example**

```
// v0    : position attribute
// o0    : output for position
// c0-c3 : modelview matrix
// c4-c7 : projection matrix
// c8    : (1 - epsilon, 1, any, any)
m4x4    r0, v0, c0              // modelview transformation
m4x4    r1, r0, c4              // projection transformation
mul     r2.xy, -r1.w, c8.xy     // r2.x = -w * (1-epsilon), r2.y = -w
cmp     2, 4, r1.xx, r2.xy      //
ifc     1, 1, 1                 // if ((x < -w * (1-epsilon)) && (x > -w))
    mov     r1.x, -r1.w         // x = -w;
endif
mov     o0, r1
```

# 15.10. Changing the Priority of Operations Within a Driver

The CPU in the 3DS has two cores, one dedicated to running applications, and the other dedicated to controlling the processes of the system's devices.

The core for the system controls multiple devices, and GPU processing has a high priority among these devices. So heavy graphics processing can affect processing for other devices. In such cases, you can use the `nngxSetInternalDriverPrioMode()` function to set the GPU to a lower priority and minimize the effect on other devices.

**Code 15-5. Changing the Priority of GPU Driver Operations**

```
void nngxSetInternalDriverPrioMode(nngxInternalDriverPrioMode mode);
```

Select from the following values for the *mode* parameter.

**Table 15-6. Internal Operation Priorities**

| Definition | Description |
|---|---|
| NN_GX_INTERNAL_DRIVER_PRIO_MODE_HIGH | High priority (default) |
| NN_GX_INTERNAL_DRIVER_PRIO_MODE_LOW | Low priority |

Note that lowering the priority for the GPU reduces the performance effect on other devices, but also reduces graphics performance.

# 15.11. Functions That Allocate Internal Buffers Within the Library

The library implicitly allocates internal buffers for some of the `gl` and `nngx()` functions.

## 15.11.1. nngxValidateState, glDraw*

If a reference table (LUT) is being used, the library allocates an internal buffer.

When `glTexture1D` is called and the library is notified that a reference table will be used, the next time `nngxValidateState` or `glDraw*` is executed, an intermediate buffer is allocated for loading the reference table. However, when a 3D command is generated directly, and a buffer for the reference table is specified, no intermediate buffer is allocated.

Note that the allocated region is freed by `glDeleteTexture` or `nngxFinalize`.

## 15.11.2. Command Lists, Display Lists, Textures, and Similar Objects

Functions for allocating command lists, display lists, textures, and other objects allocate internal buffers for each object. These buffers are maintained until the corresponding object has been

destroyed using `nngxDelete*` or `glDelete*`.

This applies to the following functions.

> `nngxGenCmdlists`, `nngxGenDisplaybuffers`, `glCreateProgram`, `glCreateShader`, `glGenBuffers`, `glGenRenderbuffers`, `glGenTextures`, and others.

## 15.12. Analyzing Causes of GPU Hangs

The data returned from a call to the `nngxGetCmdlistParameteri()` function when passing `NN_GX_CMDLIST_HW_STATE` for *`pname`* includes a number of bits that indicate whether the hardware is busy. This data can be helpful in analyzing the cause of problems in the operation of the hardware, such as when the GPU hangs.

When the hardware is malfunctioning, the likely cause is modules that are stuck in the busy state. For modules that work in sequence, such as the triangle setup > rasterization > texture unit modules, the busy signal propagates from the last module through to the first module in the chain. When a sequence of modules is busy, the last module in the chain is the most likely cause. However, the per-fragment operation module indicated by bit 6 in the returned data can be stuck in the busy state due to invalid data from a previous module, so the most likely cause could also be an earlier module.

Roughly speaking, busy signals propagate from rasterizing and pixel processing marked by bits 0 through 7, or from geometry processing marked by bits 8 through 16.

Rasterizing and pixel processing covers the sequence of modules of triangle setup > rasterization > texture unit > fragment lighting > texture combiner > per-fragment operation, with busy signals from modules later in the chain propagating to modules earlier in the chain. In other words, busy signals propagate in order from bit 5 to bit 0.

Bit 6 is also a per-fragment operation module busy signal. However, although this does propagate to bits 0 and 1, it does not propagate to bits 2, 3, or 4.

Bit 7 indicates a busy signal from the early depth test module, which occurs when the system is waiting for the early depth buffer to clear (GPU built-in memory). This busy signal does not propagate to other modules.

A busy signal in the triangle setup module does not propagate to earlier modules (the vertex cache or geometry generator). In other words, no busy signals propagate between rasterizing and pixel processing and geometry processing.

Geometry processing takes place as follows. Vertex input process module (which loads command buffers and vertex arrays) ⇒ vertex processor ⇒ post vertex cache (in that order). Busy signals from modules later in the chain propagate to modules earlier in the chain. In other words, busy signals propagate in this order: bit 16 ⇒ (bit 11, bit 12, bit 13, bit 14) ⇒ bit 8 ⇒ bit 9. Bits 11, 12, 13, and 14 correspond to the busy states of vertex processors 0, 1, 2, and 3, but because each vertex processor is allocated in parallel with the vertex loading module and post-vertex cache, a busy signal from the post-vertex cache propagates to one or more of the four vertex processors. The signal might not propagate to all four of the vertex processors.

This description applies to the situation when the geometry shader is disabled. When it is enabled, vertex processor 0, which is the geometry shader processor, comes after the post-vertex cache. In this case, a busy signal from the geometry shader processor propagates to the post-vertex cache, but it does not propagate to any earlier modules. In other words, a busy signal would propagate in this order: bit 11 > bit 16. A busy signal arising from the post-vertex cache does propagate to earlier modules (vertex processors 1, 2, and 3). In other words, a busy signal would propagate in this order: bit 11 ⇒ bit 16, and bit 16 ⇒ (bit 12, bit 13, bit 14) ⇒ bit 8 ⇒ bit 9.

The post-vertex cache, indicated by bit 16, outputs a busy signal when it is filled to capacity with vertex data. If the cache cannot output this data to the next module for some reason, such as when

the next module is not responding, vertex data fills the post-vertex cache to capacity. If the geometry shader is disabled, the next module is the triangle setup module. If the geometry shader is enabled, the next module is the geometry processor (vertex processor 0).

The cause of the GPU hanging when the texture unit indicated by bit 2 is busy, can be attributed to a bug in the hardware that occurs when simultaneously using textures, both in and not in, a 4-bit format as a multitexture.

If the GPU hangs due to an incorrect load array setting (the vertex attribute data load unit to the GPU), bit 8 enters a busy state.

If the GPU hangs due to the vertex shader's output of NaN (or the geometry shader's output pf NaN when the geometry shader is being used), the rasterization module and the triangle setup (bit 0 and bit 1) enter a busy state.

**Note:** For more information about the `nngxGetCmdlistParamerteri()` function and the bits obtained with `NN_GX_CMDLIST_HW_STATE`, see 4.1.10. Getting Command List Parameters.

## 15.12.1. Hardware State When the GPU Hangs

The following table provides examples of the hardware state obtained for `NN_GX_CMDLIST_HW_STATE` when the GPU hangs, and the related cause.

**Table 15-7. Hardware States and Causes of GPU Hangs**

| Hardware State | Cause of GPU Hang |
|---|---|
| 0x00011303<br>0x00011F03<br>0x00012F03 | CPU destroyed content of vertex buffer while GPU was operating. |
| 0x00010103<br>0x0001011B<br>0x00014303 | GPU bug caused content of vertex buffer in VRAM to be discarded in the middle of rendering.<br>For information, see 15.9. Lines Are Rendered in Error and the Region Following the Framebuffer Is Corrupted. |
| 0x00010107<br>0x00011307<br>0x00012307 | Hang due to multitexture hardware bug.<br>Conflict with texture addressing of 128-byte alignment.<br>For information, see 7.3.1. Formats With 4-Bit Components. |
| 0x00000100 | Bit 8 in both PICA registers `0x0229` and `0x0253` was not set correctly.<br>These registers must be set again when rendering using a GL function after rendering using a non-GL function.<br>Unused elements of the load array (bits 31 to 28 of register `0x0205 + n * 3`) are not set to `0`. |
| 0x00007300 | PICA register `0x0289` was set to use the geometry shader, but the system instead executed a vertex shader that does not use the geometry shader. |
| 0x00000000 | Related register for the command buffer address jump function was not set properly.<br>While in standby for command request execution, `nngxFlush3DCommandPartially` was called prior to completing configuration of the related register of the address jump function. |
| 0x00000001<br>0x00000002<br>0x00000003 | NaN was output from the vertex shader. |

**Note:** For more information about how to set a PICA register, see 3DS Programming Manual: Advanced Graphics.

## 15.13. Effect of Vertex Attribute Combinations on Vertex Data Transfer Speed

When a vertex buffer is used, the vertex attribute data type and data size combination (the *type* and *size* parameters in the `glVertexAttribPointer()` function) affects the speed of vertex data transfer.

Vertex attribute data stored in the vertex buffer is grouped together as one or multiple vertex attributes and loaded to the GPU. The load array is the unit used in loading the vertex data.

> **Note:** For more information about load arrays, see *3DS Programming Manual: Advanced Graphics*.

When the GPU transfers load arrays, it determines whether to perform a read-ahead transfer based on the combination of vertex attribute data types and data sizes comprising the load array. If a read-ahead transfer can be performed, the vertex data transfer speed is faster.

Read-ahead transfer is performed when data meets the requirements of the conditional equation shown below.

> (Attribute Number Types Other Than `GL_FLOAT` + Attribute Number Whose Data Size Is 1) <= (`GL_FLOAT` Type Attribute Number Whose Data Size Is 4 + `GL_FLOAT` Type Attribute Number Whose Data Size Is 3 / 2)

The data size of Attribute Number Types Other Than `GL_FLOAT` and the data type of Attribute Number Whose Data Size Is 1 are arbitrary. Vertex attributes applying multiple conditions are counted according to each of those conditions. For instance, for `GL_BYTE` type vertex attributes whose data size is 1, both the attribute type other than `GL_FLOAT` and the attribute number whose data size is 1 are counted.

If the conditions for read-ahead transfer are matched, transfer speed depends on the data volume of load arrays. The smaller the data volume, the faster the transfer speed. If the volume of vertex data is the same, transfer speed depends on the number of attributes included in the load array. The fewer the load array attributes, the faster the transfer speed.

## 15.14. Vertex Array Address Alignment

Efficiency of vertex array transfer processing can be improved by keeping vertex array address alignment to 32 bytes when rendering with the use of a vertex buffer. The vertex array address is a value comprised of the vertex buffer address and an offset specified by the `glVertexAttribPointer()` function (a value specified in *ptr*).

The extent to which speed is improved in comparison to a vertex array address whose alignment has not been kept within 32 bytes depends on the vertex attribute type, size, location of vertex array storage, and the content of the vertex index. There is no guarantee that this method will be effective. In addition, even if transfer processing performance improves, it will not necessarily improve performance of the overall system unless vertex array transfer processing is causing a noticeable bottleneck.

# 15.15. GPU Hangs When Multitexture Is Used

**Note:** On SNAKE hardware, the GPU does not hang when using multitextures. However, note that hangs can still occur when the application is running on CTR.

The GPU may experience a hang when both of the following conditions are met.

- Multiple textures are used.
- There is a considerable difference in performance between texture units.

Procedural texture is not included in these conditions. This phenomenon does not occur when one normal texture is used simultaneously with a procedural texture.

This phenomenon can be avoided by taking the following steps.

- Use only one texture.
- For all textures used, apply the `GL_XXX_MIPMAP_LINEAR` setting (the setting for trilinear filter use) to the `GL_TEXTURE_MIN_FILTER` texture parameter.
  This parameter must be set even for textures for which there actually is no mipmap.

In addition, the following methods are recommended as a means of mitigating this phenomenon.

- Set the `GL_TEXTURE_MIN_FILTER` texture parameter to the `GL_XXX_MIPMAP_LINEAR` setting (the setting for trilinear filter use) for a portion of the textures to be used.
  The problem can be avoided completely by setting all textures to the trilinear filter setting, but the phenomenon's occurrence can be mitigated by using the setting on just a portion of the textures.
- Place all textures to be used at the same time in the same VRAM.
- Reduce the number of textures used.

Because the occurrence of this phenomenon is dependent on timing, changing texture settings as listed below can also avoid the problem much of the time. However, in some cases, the frequency of occurrence could actually worsen.

- Change the size of textures.
- Change texture formats.
- Change the filter mode of the textures.
- Change the storage locations of textures, switching between VRAM-A, VRAM-B, and FCRAM (device memory).

For a description of determining whether this phenomenon is the cause of a hang, see 15.12. Analyzing Causes of GPU Hangs.

However, note that the same type of hangs can also be caused by being in conflict with restrictions governing the storage location of the 4-bit texture format, making it difficult to determine the cause for certain.

# 15.16. GL_INTERPOLATE Calculations of the Texture

## Combiner

The texture combiner's GL_INTERPOLATE expression is `src0 * src2 + src1 * (1 – src2)`. If `src2` is 1 or `0`, you would expect the result of the calculation to be the same value as either `src0` or `src1`. However, because of how the specifications have been implemented, if `src2` is 0 and `src0` < `src1`, the output result is not the value of `src1` as you would expect, but rather something that is one unit less bright than `src1`.

To avoid this problem, you must combine GL_MODULATE and GL_MULT_ADD_DMP and perform the calculation in a two-stage combiner. Alternatively, if you change the `src2` operand to `GL_ONE_MINUS_*` and switch `src0` and `src1`, you may be able to reduce the fragments for which this problem arises.

## 15.17. When Render Results Are a Complete Mismatch for Polygons With the Same Vertex Coordinates

Even when you render polygons that have exactly the same vertex coordinates, the various attribute values for the fragments might not match at all.

This phenomenon occurs as a result of calculation errors from fragment interpolation calculations that arise when the order in which the vertices were entered into the rasterization module was different. It will not occur when the input order for the vertices is exactly the same but if you render using the `glDrawElements()` function with GL_TRIANGLES as a parameter, that input order can change internally as a result of the relationship with the vertex indices for the immediately prior polygon. To ensure your ability to render multiple polygons with completely matching fragments, you must use the same vertex indices when rendering, including those for the polygons that precede and follow your desired polygons.

CONFIDENTIAL

# Revision History

## Version 1.6 (2016-06-24)

### Changes

- 4.1.4. Running Command List Objects
    - Added that the accumulation of commands in an executing command list must occur in the application core.

## Version 1.5 (2016-05-10)

### Changes

- 4. Command Lists
    - Added that the function issuing command requests can only be called in Core 0.

## Version 1.4 (2015-11-05)

### Additions

- 3.8. Specifying Display Portions
- 7.5. Copying From the Framebuffer
- 12.4.5.2. Silhouette Shadow Artifacts
- 15.17. When Render Results Are a Complete Mismatch for Polygons With the Same Vertex Coordinates

### Changes

- 1. Overview
    - Added notes to refer to the API Reference for alignment definition names.
- 3. LCD
    - Consolidated Path of the Rendering Results Prior to Being Displayed on the LCDs as one section.
- 3.3. Allocating Buffers
    - Added a figure (Configuration of the Framebuffer and the Display Buffer).
- 4.1.7.1. Flushing the Accumulated 3D Command Buffer
    - Fixed the function that reflects the cache content from `nngxUpdateBuffer()` to `nngxUpdateBufferLight()`.
    - Fixed the error value from `GL_ERROR_80E0_DMP` to `GL_ERROR_80B0_DMP`.

- 4.1.19. Adding a Memory Fill Command Request
    - Fixed the A bit in `GL_RGB5_A1` for the Fill Pattern by Render Buffer Format table.
- 7.9. Texture Collections
    - Added a description of texture collections.
- 9.3.8. Open Edges
    - Added a figure (Sample Indices for Open Edges).
- 9.4.6. Subdivision Patch Indices
    - Added samples for subdivision patch indices.
- 9.5.6. Subdivision Patch Indices
    - Added a figure (Sample Subdivision Patch Indices).
- 9.6.2. Reserved Uniform
    - Clarified the shader file to use.
- 10. Rasterization
    - Added a figure (Process From the Vertex and Geometry Shaders to Rasterization).
- 12.2.10. Creating and Specifying Lookup Tables
    - Changed `dmp_LightEnv.samplerRR` to `dmp_FragmentMaterial.samplerRR` in the Specifying the Reflection Lookup Table (for the R Component) code sample.
- 12.6.3. Shading Pass
    - Noted the uniform name used when enable the fog feature's gas mode.
- 13. Per-Fragment Operations
    - Added a figure (Per-Fragment Operations Process in Standard Mode).
- 13.2.1. How to Use
    - Added that there is no difference in performance whether either the stencil test, the depth test, or both are enabled.
- 13.3.1. How to Use
    - Added that there is no difference in performance whether either the stencil test, the depth test, or both are enabled.

- 15.1. glFinish and glFlush Functions
    - Deleted the description that you can explicitly instruct the library to execute all 3D commands.
- 15.2. Differences From OpenGL ES
    - Added the `glGetShaderSource()` function to the Differences by Function table.

### Deletions

- Path of the Rendering Results Prior to Being Displayed on the LCDs
    - Consolidated into 3. LCD.

## Version 1.3 (2015-04-28)

### Changes

- 11.1.6. Texture Cache
    - Added information about texture cache.
- 15.15. GPU Hangs When Multitexture Is Used
    - Added information about using trilinear filtering as a workaround for GPU hangs.

## Version 1.2 (2015-01-15)

### Changes

- 4.1.16. Adding a Block-to-Linear Image Conversion and Transfer Command Request
    - Added a note about the image size to specify in a transfer request.
    - Added descriptions of the errors generated by the `nngxAddB2LTransferCommand()` function.

## Version 1.1 (2014-11-10)

### Changes

- 3.5. Copying From the Color Buffer to the Display Buffer
    - Added a description of the minimum value for the copy region in calls to `nngxTransferRenderImage` and the error that occurs if a smaller value is specified.

## Version 1.0 (2014-09-04)

### Additions and Changes

- Initial version.

# 3DS Programming Manual: Advanced Graphics

Version 1.3

**Nintendo Confidential**

This document contains confidential and proprietary information of Nintendo, and is protected under confidentiality agreements as well as the intellectual property laws of the United States and of other countries. No part of this document may be released, distributed, transmitted, or reproduced in any form , including by any electronic or mechanical means and by including within information storage and retrieval systems, without written permission from Nintendo.

---

©2015 Nintendo. All rights reserved.

All company and product names in this document are the trademarks or registered trademarks of their respective companies.

CTR-06-0076-002-D

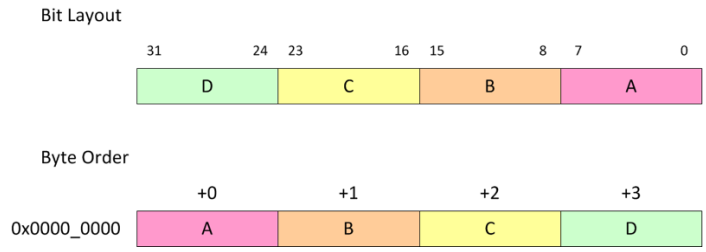
# 1. Introduction

This document describes programming using the 3DS system's graphics features. It also provides sample implementations of various lighting models and features that require special configuration, such as block modes and state caches. This document assumes that you have first read the 3DS Programming Manual: System and 3DS Programming Manual: Basic Graphics.

## 1.1. Bit Layout Notation

The 3DS system is little-endian, so the byte order in memory and the bit layout order do not correspond. The figure below compares the bit layout of 32-bit data with its byte order in memory.

**Figure 1-1. Bit Layout and Byte Order**






# 2. Block Mode Settings

The 3DS render buffer can be set to block-8 mode or block-32 mode. The render buffer must be in block-32 mode when using early depth tests (described later). If you are not using early depth tests, leave the render buffer in the initial default setting of block-8 mode.

Use the `glRenderBlockModeDMP()` function to set the block mode.

**Code 2-1. glRenderBlockModeDMP Function**

```
void glRenderBlockModeDMP(GLenum mode);
```

The *mode* parameter can take one of the following values.

**Table 2-1. Block Mode Setting Values**

| Setting Value | Block Mode |
|---|---|
| `GL_RENDER_BLOCK8_MODE_DMP` | Block-8 mode (default) |
| `GL_RENDER_BLOCK32_MODE_DMP` | Block-32 mode |

Unlike block-8 mode, block-32 mode has the following restrictions.

- The block size is changed to 32, so the render buffer and display buffer width and height must both be in multiples of 32 pixels. Due to this restriction, it is not possible to allocate a render buffer or display buffer of the exact size of the LCD screen.
- The `glCopyTexImage2D()` and `glCopyTexSubImage2D()` functions do not correctly copy images to textures.
- Rendered results will not be correct when specifying a texture in the framebuffer (render to texture). Due to this restriction, it is not possible to carry out any processing that requires rendering to textures, such as rendering shadows or gases.

Rendered results are not guaranteed if you render while changing the block mode. All calls to the `glDrawElements()`, `glDrawArrays()`, and `glReadPixels()` functions on the same render buffer must be run in the same block mode. Call the `glGetIntegerv()` function and pass `GL_RENDER_BLOCKMODE_DMP` as an argument to get the current block mode.

# 2.1. Early Depth Test

The early depth test is a depth test that runs at the same time as rasterization. It can be used to discard unnecessary fragments at an earlier stage than the standard depth test. To use this feature, the render buffer must be in block-32 mode.

The early depth test is run using the early depth buffer. The early depth buffer treats a 32x32-pixel square as one block, with each block represented by one depth value. This representative value is stored with 12-bit precision, and the early depth value for the corresponding block is updated when depth values are written to the standard depth buffer. There is no need to allocate memory for the early depth buffer, and the early depth buffer supports depth buffers up to 1024×1024 pixels in size. The test compares the depth value of the polygon currently being rendered, as calculated at rasterization time, against the block's representative depth value (which is either the minimum or maximum depth value). If a block passes the early depth test, all of its fragments are passed along to the next process. If it fails the test, all fragments in the block are discarded.

The early depth test can be used to reduce the number of fragments sent through the fragment pipeline. However, the early depth test is less precise than the standard depth test, so it sometimes yields different results. Note that performance might improve less than expected, or fragments that are normally rendered might instead be discarded. Any fragments that are normally discarded (but are instead passed) are discarded by the standard depth test, so false passes do not have much effect on performance. However, false failures that discard fragments that are normally passed on for rendering may lead to display problems.

The early depth test is intended for use as a depth test that runs early in the pipeline, and it must be disabled (along with the standard depth test), whenever the standard depth test is disabled. If the early depth test alone is enabled, fragments that are normally rendered might be left out.

## 2.1.1. Test Control

Much as with the standard depth test, use the `glEnable()` and `glDisable()` functions to enable and disable the early depth test. Use the `glIsEnabled()` function to determine whether the early depth test is enabled. Pass `GL_EARLY_DEPTH_TEST_DMP` as the argument for the functions.

**Code 2-2. Early Depth Test Control**

```
glEnable(GL_EARLY_DEPTH_TEST_DMP);
glDisable(GL_EARLY_DEPTH_TEST_DMP);
glIsEnabled(GL_EARLY_DEPTH_TEST_DMP);
```

## 2.1.2. Clearing the Buffer

The early depth buffer is independent of the standard depth buffer. It can be cleared independently, but make sure that you also clear the early depth buffer whenever you clear the standard depth buffer.

To clear the early depth buffer, call the `glClear()` function and pass `GL_EARLY_DEPTH_BUFFER_BIT_DMP` for the *`mask`* parameter. Use the `glClearEarlyDepthDMP()` function to set the value used to clear the buffer.

**Code 2-3. Setting the Value Used to Clear the Early Depth Buffer**

```
void glClearEarlyDepthDMP(GLuint depth);
```

The values used to clear the early depth buffer and the standard depth buffer are very similar, but whereas a `GLfloat` is used to clear the standard depth buffer, you must specify a positive integer for the early depth buffer. Use the following equation to calculate the value to specify.

$$Depth_{Early} = Depth \times 0xFFFFFF + offset \quad (0.0 \le Depth \le 1.0)$$

Because the early depth test is less precise than the standard depth test, we recommend using a positive value of `0x1000` or greater for *offset* when using `GL_LESS` or `GL_LEQUAL` modes, and a value of `–0x1000` or lower when using `GL_GREATER` or `GL_GEQUAL` modes. Valid clear values for the early depth buffer lie in the range from `0x000000` through `0xFFFFFF`, so make sure to clamp to within this range if the results after adding *offset* ever fall outside these bounds.

To get the configured clear value, call the `glGetIntegerv()` function and pass `GL_EARLY_DEPTH_CLEAR_VALUE_DMP` for the *pname* parameter.

## 2.1.3. Depth Buffer Update Flag

The early depth buffer contains early depth values in blocks of 32×32 pixels, in addition to update flags of the standard depth buffer (the depth buffer update flags) in 4×4 pixel units. This flag is set to 1 when even one pixel of the corresponding 4×4 pixel block in the depth buffer is changed as the result of the standard depth test. The early depth value held in the early depth buffer block that contains that pixel is then updated. When this flag is set to 1, the flag does not change, even if pixels in the corresponding block are updated in the depth buffer, and the early depth values are then also not updated. However, when the depth buffer is updated for pixels for which the corresponding flag is still `0`, even when included in the same early depth buffer block, the flag is set to 1 and the early depth value is updated. The depth buffer update flag is only cleared to `0` when the early depth buffer is cleared.

The early depth test results are strongly dependent on the rendering order of the model, due to this controlling effect of the depth buffer update flag. To increase the effect of the early depth test, set the comparison function to `GL_LESS` or `GL_LEQUAL` to render a model from the foreground toward the back, and set to `GL_GREATER` or `GL_GEQUAL` to render from the back toward the foreground.

For example, when the comparison function is set to `GL_LESS`, rendering model objects farther toward the background first means that objects more in the foreground that are rendered later will not have their early depth buffers updated, causing an increase in the number of fragments that pass the early depth test and resulting in less of an effect from the early depth test. Conversely, rendering objects more in the foreground first results in an increase in the number of fragments discarded by the early depth test, and a greater effect from the test.

When setting as above and rendering the background first, the early depth test has no effect at all. In such cases, disable the early depth test just when rendering the background and then enable it when rendering other models to get at least some effect from the early depth test.

**Note:** The early depth value is not updated for pixels that have already been rendered once because of the update flag. You must clear the early depth buffer each frame to set the update flag to 0, even though you change the comparison function between even and odd frames to avoid clearing the depth buffer.

## 2.1.4. Setting the Comparison Function

The comparison function to use for early depth tests must be configured separately from the standard depth test's comparison function. Use the `glEarlyDepthFuncDMP()` function to set the comparison function.

**Code 2-4. Setting the Early Depth Test Comparison Function**

```
void glEarlyDepthFuncDMP(GLenum func);
```

The *`func`* parameter specifies the comparison function. The four comparison functions that can be set are `GL_LESS`, `GL_LEQUAL`, `GL_GREATER`, and `GL_GEQUAL`. No other comparison function can be set. Setting different comparison functions for the early depth test and standard depth test can cause inconsistencies in the pass/failure results of the tests.

Due to restrictions on the comparison function for the early depth test, it cannot be changed easily. If you change the depth test comparison function during rendering of a single scene, the early depth test can no longer be used until the early depth buffer is cleared.

However, this buffer is the only one that needs to be cleared to use the test again. To get the currently set comparison function, call the `glGetIntegerv()` function and pass `GL_EARLY_DEPTH_FUNC_DMP` for the *`pname`* parameter. `GL_LESS` is the default setting.

If the early depth test is enabled, the corresponding early depth buffer value is updated to the maximum or minimum depth value of the fragments in that block whenever values are written to the standard depth buffer. The early depth buffer is updated to the maximum value when the early depth test comparison function is set to the `GL_LESS` or `GL_LEQUAL` mode, and to the minimum value when set to the `GL_GREATER` or `GL_GEQUAL` mode. The early depth buffer contents cannot be directly read or set from outside.

### 2.1.4.1. Changing the Comparison Function

Regardless of whether you change the early depth test comparison function during the rendering of a single frame, there are several restrictions you must observe whenever you change this comparison function.

- The comparison function for the early depth test must be the same as for the standard depth test.
- After an early depth test is set to a particular comparison function, this comparison function cannot be changed until the early depth buffer is cleared.
- If the standard depth test comparison function is changed for a particular render buffer and something more is rendered to that buffer, the early depth test cannot be enabled until the early depth buffer is cleared.
- If the standard depth test comparison function is changed to one that cannot be used for the early depth test, the early depth test must be disabled.

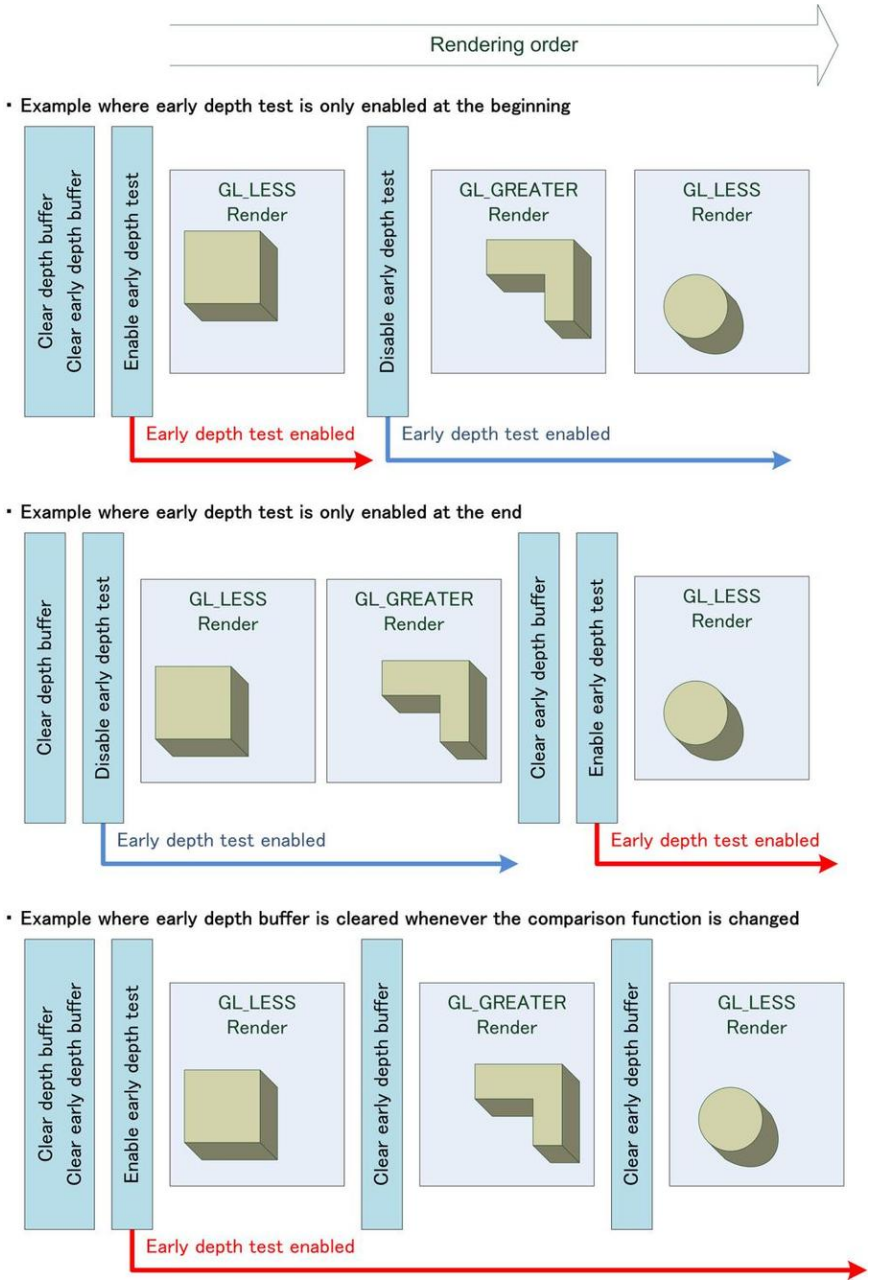
Failing to comply with these restrictions may lead to the wrong fragments being discarded, resulting in incorrectly rendered results.

Some ways of avoiding these restrictions include: only enabling the early depth test in those locations where you want to improve performance, or always clearing the early depth buffer

whenever you change the comparison function. However, clearing the early depth buffer too often can cause an increase in false passes, and prevent any improvements in performance.

When you clear the standard depth buffer and early depth buffer at different times (such as when you change the comparison function within a single frame), use either the minimum (`0x000000`) or maximum (`0xFFFFFF`) clear values to help avoid false test results.

**Figure 2-1. Example of Workaround for Comparison Function Change Restrictions**


Rendering order
• Example where early depth test is only enabled at the beginning
Clear depth buffer
Clear early depth buffer
Enable early depth test
GL_LESS Render
Disable early depth test
GL_GREATER Render
GL_LESS Render
Early depth test enabled
Early depth test enabled
• Example where early depth test is only enabled at the end
Clear depth buffer
Disable early depth test
GL_LESS Render
GL_GREATER Render
Clear early depth buffer
Enable early depth test
GL_LESS Render
Early depth test enabled
Early depth test enabled
• Example where early depth buffer is cleared whenever the comparison function is changed
Clear depth buffer
Clear early depth buffer
Enable early depth test
GL_LESS Render
Clear early depth buffer
GL_GREATER Render
Clear early depth buffer
GL_LESS Render
Early depth test enabled

## 2.1.5. Problems When Using Viewports With an Offset

There is a hardware bug with the early depth test that prevents the viewport offset from being applied to the coordinates that are read from the early depth buffer. As a result, the early depth test is not run properly when the viewport offset (the *x* and *y* parameters to the `glViewport()` function) is not (0, 0).

Because the viewport offset is applied to the coordinates that are updated in the early depth buffer, the depth value of pixels with the viewport offset applied is compared with the depth values in the early depth buffer without the offset applied. This causes pixels that normally pass the early depth test to fail, and prevents pixels that are normally rendered from being rendered. When a pixel fails the early depth test, its entire 8×8 pixel block is mistakenly discarded.

The following are two ways to work around this issue.

- Set a viewport offset other than (0, 0) and disable the early depth test. The early depth test is less effective if it switches between being enabled and disabled while a single scene is rendered, but the rendered results are the same because the normal depth test discards any pixels that are normally discarded.
- Set the viewport offset to (0, 0) and then use a modelview transformation to shift the render region and cut off unnecessary areas with the scissor test.



# 3. Using Data Located in Main Memory

Vertex buffers and texture images can be located not just in VRAM but also in regions allocated in main memory. "Main memory" in this case refers more specifically to "device memory," which is the portion of main memory where the operating system guarantees the integrity of the addresses accessed by peripheral devices. When specifying graphics data such as vertex buffers and textures in main memory, if you do not use the GL library to create copies of the data in main memory, you must store the data in device memory.

**Note:** The alignment of addresses differs depending on buffer type (in other words, depending on the purpose of the memory region). For more information, see the Differences in Alignment Between Buffer Types table in 3.2.1.1. Allocator of the *3DS Programming Manual: Basic Graphics*. For more information about device memory, see the *3DS Programming Manual: System*.

The GPU can directly reference data located in main memory, but it works differently depending on the settings specified when that region was allocated. Also, the access speed of main memory is slower than VRAM, so data location can have an effect on performance. For example, if you allocated memory for vertex buffers in both VRAM and main memory, the access speed for the buffers in VRAM would decrease to match the access speed for the buffers in main memory.

Pass a bitwise OR of certain specific flag values in the *`target`* parameter of the `glTexImage2D`, `glCompressedTexImage2D`, `glCopyTexImage2D`, and `glBufferData()` functions to configure the GPU access targets and whether to create a copy in main memory when the memory region is allocated.

**Table 3-1. GPU Access Target Flags**

| Flag | Access Target |
|---|---|
| NN_GX_MEM_VRAMA | GPU accesses VRAM-A. |

| | |
|---|---|
| NN_GX_MEM_VRAMB | GPU accesses VRAM-B. |
| NN_GX_MEM_FCRAM | GPU accesses main memory. |

**Table 3-2. Copy Creation Flags**

| Flag | Copy Creation |
|---|---|
| GL_COPY_FCRAM_DMP | Create copy in main memory. |
| GL_NO_COPY_FCRAM_DMP | Do not create copy in main memory. |

These flags produce the following four setting combinations.

1. Access Main Memory and Create a Copy in Main Memory
2. Access Main Memory and Do Not Create a Copy in Main Memory
3. Access VRAM-A (or B) and Create a Copy in Main Memory
4. Access VRAM-A (or B) and Do Not Create a Copy in Main Memory

Combination (1) is the default for the `glTexImage2D()`, `glCompressedTexImage2D()`, and `glBufferData()` functions.

For just the `glTexImage2D()` function, passing NULL in the *`pixels`* parameter sets VRAM-B as the access target for combination (4).

The arguments to the `glCopyTexImage2D()` function can only specify the GPU access target. The function allocates a region in the specified memory, and then transfers the color buffer contents via DMA. NN_GX_MEM_FCRAM (access main memory) is the default.

The `glBufferSubData()` function inherits the configured GPU access target and copy creation flags from the `glBufferData()` function.

## 3.1. Access Main Memory and Create a Copy in Main Memory

This setting combination applies when you specify the bitwise OR of NN_GX_MEM_FCRAM and GL_COPY_FCRAM_DMP during a function call.

The function carries out the following operations.

- The function allocates a memory region in main memory.
- The CPU copies data to the allocated region.
- The GPU directly accesses main memory (the region to which the data was copied).

The region storing the data can be released immediately after the function completes (this is immediately after data is copied by the CPU). If NULL is specified as the data address, a region is allocated but data is not copied.

## 3.2. Access Main Memory and Do Not Create a Copy in Main

This setting combination applies when you specify the bitwise OR of NN_GX_MEM_FCRAM and GL_NO_COPY_FCRAM_DMP during a function call.

The function carries out the following operations.

- The GPU directly accesses main memory (the region allocated by the application).

The region storing the data cannot be released until rendering completes. A GL_INVALID_OPERATION error occurs when NULL was specified as the data address and when textures were loaded that are not in PICA native format.

With this setting combination, the application manages the memory for the vertex buffer allocated by the glBufferData() function, so the glBufferSubData() function does not update the data of the partial regions but instead only handles cache flushing of the partial regions. Updating the partial regions is the responsibility of the application. A GL_INVALID_VALUE error occurs if the value passed in the glBufferSubData() function's *data* parameter is not a sum of *offset* and the source buffer address specified in the glBufferData() function.

When using the GPU to render data updated dynamically by the CPU and located in main memory, but without calling any GL function, you must call the nngxUpdateBufferLight() function.

## 3.3. Access VRAM-A (or B) and Create a Copy in Main Memory

This setting combination applies when you specify the bitwise OR of NN_GX_MEM_VRAMA (or NN_GX_MEM_VRAMB) and GL_COPY_FCRAM_DMP during a function call.

The function carries out the following operations.

- The function allocates memory regions in both main memory and VRAM.
- The CPU copies data to the allocated region.
- Data is sent via DMA transfer from main memory (the copy region) to VRAM.
- The GPU accesses VRAM.

The region storing the data can be released immediately after the function completes (this is immediately after data is copied by the CPU). A GL_INVALID_OPERATION error occurs when NULL was specified as the data address.

## 3.4. Access VRAM-A (or B) and Do Not Create a Copy in Main Memory

This setting combination applies when you specify the bitwise OR of NN_GX_MEM_VRAMA (or NN_GX_MEM_VRAMB) and GL_NO_COPY_FCRAM_DMP during a function call.

The function carries out the following operations.

- The function allocates a memory region in VRAM.
- Data is sent via DMA transfer from main memory (the region allocated by the application) to VRAM.
- The GPU accesses VRAM.

The region storing the data cannot be released until the DMA-transfer copy operation completes (that is, until completion of the DMA transfer command request is confirmed). If `NULL` is specified as the data address, a region is allocated but no DMA transfer is performed. A `GL_INVALID_OPERATION` error occurs when textures were loaded that are not in PICA native format.

When you call the `glBufferSubData()` function on vertex buffers loaded with this setting combination, the application must guarantee the content of the memory region specified in *`data`* until the DMA transfer completes.

## 3.5. Effectively Using VRAM

VRAM-A and VRAM-B are accessed using separate channels. If a buffer is sometimes loaded and written at the same time, we recommend allocating that buffer in both VRAMs and using the buffer in one VRAM for loading and the other for writing.

The following combinations can cause competing access.

**Table 3-3. Combinations That Cause Competing Access**

| Combinations That Cause Competing Access | When It Occurs |
|---|---|
| Color buffer and depth (stencil) buffer | Throughout rendering |
| Color buffer and display buffer | When the `nngxTransferRenderImage()` function runs |
| Vertex buffer and index buffer | When the `glDrawElements()` function runs |

We strongly recommend allocating both the color buffer and depth (stencil) buffer in separate VRAM banks because these buffers are accessed throughout rendering.



# 4. State Caches

**Note:** Due to inferiority in features compared to command caches, state caches have been eliminated.

# 5. Lighting Model Sample Implementation

Lighting models are mathematical models that express how a three-dimensional scene is lit. They include information such as the colors of the light sources and materials, and how the light is reflected.

A typical lighting model can be broken down into several components: emissive, ambient, diffuse, and specular light.

$$reflected\_light = emissive + ambient + diffuse + specular$$

Fragment lighting outputs a primary and a secondary color. The primary color is the sum of the emissive, ambient, and diffuse light, while the secondary color is the sum of two specular lights and can be used to express more than just simple specular reflection.

## 5.1. Microfacet Reflection

Microfacets are tiny surfaces that cannot be seen with the naked eye. Assuming that a rough surface is composed of many microfacets, those microfacets whose normals coincide with the half-angle vector of the light vector and the view vector reflect light along the line of sight toward the viewer. In other words, the greater the number of microfacets there are that reflect along the line of sight, the stronger the reflected light that the viewer sees.

If you implement a distribution function that returns the percentage of microfacets whose normals coincide with the angle supplied as an argument, the amount of light reflected by a substance will be proportional to that function. This function also shows how the microfacet slopes are distributed, so it can also be called a slope distribution function.

**Code 5-1. Microfacet Reflection**

```
GLfloat ms0[] = {1.0f, 1.0f, 1.0f, 1.0f};
GLfloat ms1[] = {0.0f, 0.0f, 0.0f, 1.0f};
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.config"),
        GL_LIGHT_ENV_LAYER_CONFIG0_DMP);
glUniform4fv(glGetUniformLocation(progID, "dmp_FragmentMaterial.specular0"),
        1, ms0);
glUniform4fv(glGetUniformLocation(progID, "dmp_FragmentMaterial.specular1"),
        1, ms1);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutEnabledRefl"),
        GL_FALSE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutEnabledD0"),
        GL_TRUE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.absLutInputD0"),
        GL_TRUE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutInputD0"),
        GL_LIGHT_ENV_NH_DMP);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.clampHighlights"),
        GL_FALSE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightSource[0].geomFactor1"),
        GL_FALSE);

GLfloat LUT_D0[512];
for( i = 0; i < 256; i++) LUT_D0[i] = slopeDist((float) i / 256.0f);
for( i = 0; i < 255; i++) LUT_D0[i + 256] = LUT_D0[i + 1] - LUT_D0[i];
LUT_D0[511] = slopeDist(1.0f) - LUT_D0[255];

glBindTexture(GL_LUT_TEXTURE0_DMP, lutTexD0);
glTexImage1D(GL_LUT_TEXTURE0_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
        GL_LUMINANCEF_DMP, GL_FLOAT, LUT_D0);
glUniform1i(glGetUniformLocation(progID,
```

```
        "dmp_FragmentMaterial.samplerD0"), 0);
```

Set the slope distribution sampling values in the distribution factor 0 (D0) lookup table.

## 5.2. Blinn-Phong Model

The Blinn-Phong reflection model is a simple microfacet reflection model that uses an exponential function as its distribution function. The generalized Blinn-Phong model, in addition to the exponential function, also incorporates another distribution function that takes the dot product of the normal and the half-angle vector as an argument. The Gaussian function is one example of a possible distribution function it can use.

A simple Blinn-Phong reflection model can implement microfacet reflection using only the slopeDist distribution function, as defined below.

**Code 5-2. Distribution Function in a Simple Blinn-Phong Model**

```
float slopeDist(float NH)
{
    return powf(NH, 2.0f);
}
```

It is also possible to represent a two-layered Blinn-Phong model using the two specular lights in the secondary color. Slope distribution sampling values for the second layer are set in the distribution factor 1 (D1) lookup table, and this can be used by changing to a layer configuration that uses both D0 and D1.

**Code 5-3. Two-Layer Blinn-Phong Model**

```
GLfloat ms0[] = {1.0f, 1.0f, 1.0f, 1.0f};
GLfloat ms1[] = {1.0f, 1.0f, 1.0f, 1.0f};
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.config"),
        GL_LIGHT_ENV_LAYER_CONFIG2_DMP);
glUniform4fv(glGetUniformLocation(progID, "dmp_FragmentMaterial.specular0"),
        1, ms0);
glUniform4fv(glGetUniformLocation(progID, "dmp_FragmentMaterial.specular1"),
        1, ms1);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutEnabledRefl"),
        GL_FALSE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutEnabledD0"),
        GL_TRUE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.absLutInputD0"),
        GL_TRUE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutInputD0"),
        GL_LIGHT_ENV_NH_DMP);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutEnabledD1"),
        GL_TRUE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.absLutInputD1"),
        GL_TRUE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutInputD1"),
        GL_LIGHT_ENV_NH_DMP);

glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.clampHighlights"),
        GL_FALSE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightSource[0].geomFactor0"),
        GL_FALSE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightSource[0].geomFactor1"),
        GL_FALSE);

GLfloat LUT_D0[512], LUT_D1[512];
for( i = 0; i < 256; i++) LUT_D0[i] = slopeDistD0((float) i / 256.0f);
```

```
for( i = 0; i < 255; i++) LUT_D0[i + 256] = LUT_D0[i + 1] - LUT_D0[i];
LUT_D0[511] = slopeDistD0(1.0f) - LUT_D0[255];

for( i = 0; i < 256; i++) LUT_D1[i] = slopeDistD1((float) i / 256.0f);
for( i = 0; i < 255; i++) LUT_D1[i + 256] = LUT_D1[i + 1] - LUT_D1[i];
LUT_D1[511] = slopeDistD1(1.0f) - LUT_D1[255];

glBindTexture(GL_LUT_TEXTURE0_DMP, lutTexD0);
glTexImage1D(GL_LUT_TEXTURE0_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
        GL_LUMINANCEF_DMP, GL_FLOAT, LUT_D0);
glBindTexture(GL_LUT_TEXTURE1_DMP, lutTexD1);
glTexImage1D(GL_LUT_TEXTURE1_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
        GL_LUMINANCEF_DMP, GL_FLOAT, LUT_D1);
glUniform1i(glGetUniformLocation(progID, "dmp_FragmentMaterial.samplerD0"), 0);
glUniform1i(glGetUniformLocation(progID, "dmp_FragmentMaterial.samplerD1"), 1);
```

For more information about the Blinn-Phong model, see the following.

> Blinn, James F., Models of Light Reflection for Computer Synthesized Pictures, ACM Computer Graphics, SIGGRAPH 1977 Proceedings, 11(4), pp. 192-198

## 5.3. Cook-Torrance Model

The Cook-Torrance model more accurately represents the amount of reflected light. This model defines the amount of reflected light as proportional to a distribution function multiplied by a Fresnel factor. By representing the refraction and reflection of transmitted light as a function of the angle of incidence and the refractive index, and by considering the different absorption coefficients of different substances, it is possible for this to model the reflections of various different substances. The Fresnel factor expresses the reflection as a function of the light vector and the normal vector.

When considering microfacet Fresnel factors, the normals in question are the normals of the microfacets. Just as in microfacet reflection, these normals coincide with the half-angle vector. Consequently, when this model is applied to fragment lighting, the view vector and half-angle vector (which bisects the angle formed by the view and light vectors) are used as inputs for the Fresnel factor, and the distribution function is a Beckmann function that excludes the $m^2$ denominator. The shadows cast by the microfacets onto other microfacets must also be considered, but the proportion of the microfacet reflection that reaches the viewpoint (which depends on the microfacet angle) is already included in the secondary color calculation in the form of geometry factors. These geometry factors are similar to the Cook-Torrance geometric factors, but also take angle into account. The 3DS geometry factors approximate the Cook-Torrance geometric factors as follows.

$$GF = \max\left(0, \quad \frac{Lf_i \cdot Nf}{\left(Lf_i + Vf\right)^2}\right)$$

$Lf_i$ is the $i$ light vector, and $Nf$ and $Vf$ are the normal and view vectors.

The refractive index and absorption coefficient of the Fresnel factor depend on the color. The Fresnel factor can thus be calculated using the refractive indexes and absorption coefficients of the individual RGB components, and handled as the reflection per each component in fragment lighting.

Given this, reflections, distribution factors, and geometry factors are used to represent the Cook-Torrance model in fragment lighting. The dot product of the view vector and the half-angle vector is used as the input for reflection, and the dot product of the normal and the half-angle vector is used as the input for the distribution factors. This is possible for layer configurations 4, 5, and 7. Note that specular light 1 of the material is replaced with the reflection, and that the color is determined by specular light 1 from the light.

**Code 5-4. Cook-Torrance Model**

```
GLfloat ms[] = {0.0f, 0.0f, 0.0f, 0.0f};
GLfloat ls[] = {1.0f, 1.0f, 1.0f, 1.0f};

glUniform4fv(glGetUniformLocation(progID, "dmp_FragmentMaterial.specular0"),
        1, ms);
glUniform4fv(glGetUniformLocation(progID,
        "dmp_FragmentLightSource[0].specular1"), 1, ls);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutEnabledRefl"),
        GL_TRUE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.config"),
        GL_LIGHT_ENV_LAYER_CONFIG4_DMP);

glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutEnabledD1"),
        GL_TRUE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.absLutInputRR"),
        GL_FALSE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.absLutInputRG"),
        GL_FALSE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.absLutInputRB"),
        GL_FALSE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.absLutInputD1"),
        GL_FALSE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutInputRR"),
        GL_LIGHT_ENV_VH_DMP);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutInputRG"),
        GL_LIGHT_ENV_VH_DMP);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutInputRB"),
        GL_LIGHT_ENV_VH_DMP);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutInputD1"),
        GL_LIGHT_ENV_NH_DMP);
glUniform1f(glGetUniformLocation(progID, "dmp_LightEnv.lutScaleRR"), 2.0f);
glUniform1f(glGetUniformLocation(progID, "dmp_LightEnv.lutScaleRG"), 2.0f);
glUniform1f(glGetUniformLocation(progID, "dmp_LightEnv.lutScaleRB"), 2.0f);
glUniform1f(glGetUniformLocation(progID, "dmp_LightEnv.lutScaleD1"), 2.0f);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.clampHighlights"),
        GL_FALSE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightSource[0].geomFactor1"),
        GL_TRUE);

GLfloat LUT_RR[512], LUT_RG[512], LUT_RB[512], LUT_D1[512];
for( i = 0; i < 128; i++) LUT_RR[i] =
        nk_fresnel((float) i / 128.0f, FRESNEL_N_R, FRESNEL_K_R);
for( i = 0; i < 127; i++) LUT_RR[i + 256] = LUT_RR[i + 1] - LUT_RR[i];
LUT_RR[383] = nk_fresnel(1.0f, FRESNEL_N_R, FRESNEL_K_R) - LUT_RR[127];

for( i = 0; i < 128; i++) LUT_RG[i] =
        nk_fresnel((float) i / 128.0f, FRESNEL_N_G, FRESNEL_K_G);
for( i = 0; i < 127; i++) LUT_RG[i + 256] = LUT_RG[i + 1] - LUT_RG[i];
LUT_RG[383] = nk_fresnel(1.0f, FRESNEL_N_G, FRESNEL_K_G) - LUT_RG[127];

for( i = 0; i < 128; i++) LUT_RB[i] =
        nk_fresnel((float) i / 128.0f, FRESNEL_N_B, FRESNEL_K_B);
for( i = 0; i < 127; i++) LUT_RB[i + 256] = LUT_RB[i + 1] - LUT_RB[i];
LUT_RB[383] = nk_fresnel(1.0f, FRESNEL_N_B, FRESNEL_K_B) - LUT_RB[127];

for( i = 0; i < 128; i++) LUT_D1[i] = beckmann((float) i / 128.0f, 1.0f);
for( i = 0; i < 127; i++) LUT_D1[i + 256] = LUT_D1[i + 1] - LUT_D1[i];
LUT_D1[383] = beckmann(1.0f, 1.0f) - LUT_D1[127];

glBindTexture(GL_LUT_TEXTURE0_DMP, lutTexRR);
glTexImage1D(GL_LUT_TEXTURE0_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
        GL_LUMINANCEF_DMP, GL_FLOAT, LUT_RR);
glBindTexture(GL_LUT_TEXTURE1_DMP, lutTexRG);
glTexImage1D(GL_LUT_TEXTURE1_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
        GL_LUMINANCEF_DMP, GL_FLOAT, LUT_RG);
glBindTexture(GL_LUT_TEXTURE2_DMP, lutTexRB);
glTexImage1D(GL_LUT_TEXTURE2_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
        GL_LUMINANCEF_DMP, GL_FLOAT, LUT_RB);
glBindTexture(GL_LUT_TEXTURE3_DMP, lutTexD1);
glTexImage1D(GL_LUT_TEXTURE3_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
        GL_LUMINANCEF_DMP, GL_FLOAT, LUT_D1);
glUniform1i(glGetUniformLocation(progID, "dmp_FragmentMaterial.samplerRR"), 0);
glUniform1i(glGetUniformLocation(progID, "dmp_FragmentMaterial.samplerRG"), 1);
glUniform1i(glGetUniformLocation(progID, "dmp_FragmentMaterial.samplerRB"), 2);
glUniform1i(glGetUniformLocation(progID, "dmp_FragmentMaterial.samplerD1"), 3);
```

The `nk_fresnel()` function is a Fresnel reflection function that takes the light absorption coefficient into account, and `beckmann` is, unsurprisingly, a Beckmann function.

The `dmp_LightEnv.lutScaleXX` uniforms are used to double the lookup table outputs.

For more information about the Cook-Torrance model, see the following.

Cook, Robert L., and Torrance, Kenneth E., A Reflectance Model for Computer Graphics, ACM Computer Graphics, SIGGRAPH 1981 Proceedings, 15(4), pp. 307-316.

For more information about geometry factors, see the following.

Alan Watt, 3D Computer Graphics, 3rd edition, Addison-Wesley Publishing Ltd, Addison-Wesley Publishing Company Inc., 2000, pp. 216

For more information about the refraction indices and absorption coefficients of various substances, see the following.

Glassner, Andrew S., Principles of Digital Image Synthesis, Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1995.

## 5.4. Schlick Anisotropic Model

The lighting models introduced previously assume that all facing surfaces have the same qualities (that is, they are isotropic), such that reflections are always symmetric with respect to the normal. Conversely, a substance is described as anisotropic if its surfaces have different properties depending on their orientation, such as the way a fabric surface like velvet might reflect differently for different orientations.

The light-scattering properties of an anisotropic surface vary as a function of the direction along the surface over which you measure those properties. Schlick proposed using a distribution function made up of two functions to represent anisotropic reflection. One function depends on the normal and the half-angle vector. The other depends on the angle Φ formed by the tangent and the vector projected by the half-angle vector on the tangent plane, as represented by the following equations.

$$D = Z(N \cdot H) \times A(\cos(\Phi))$$

$$Z(t) = \frac{r}{(1 + rt^2 - t^2)^2}$$

$$A(\omega) = \sqrt{\frac{p}{p^2 - p^2\omega^2 + \omega^2}}$$

*r* is a parameter characterizing the roughness of the surface, and *p* is a parameter characterizing the anisotropy of the surface.

The geometric factor in this model differs from that used in the Cook-Torrance model, but it has properties similar to the geometry factors used in fragment lighting.

**Code 5-5. Schlick Anisotropic Model**

```
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutEnabledRefl"),
        GL_TRUE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.config"),
        GL_LIGHT_ENV_LAYER_CONFIG7_DMP);
```

```
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutEnabledD1"),
        GL_TRUE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.absLutInputRR"),
        GL_TRUE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.absLutInputRG"),
        GL_TRUE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.absLutInputRB"),
        GL_TRUE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.absLutInputD1"),
        GL_TRUE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutInputRR"),
        GL_LIGHT_ENV_NH_DMP);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutInputRG"),
        GL_LIGHT_ENV_NH_DMP);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutInputRB"),
        GL_LIGHT_ENV_NH_DMP);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutInputD1"),
        GL_LIGHT_ENV_CP_DMP);
glUniform1f(glGetUniformLocation(progID, "dmp_LightEnv.lutScaleRR"), 1.0f);
glUniform1f(glGetUniformLocation(progID, "dmp_LightEnv.lutScaleRG"), 1.0f);
glUniform1f(glGetUniformLocation(progID, "dmp_LightEnv.lutScaleRB"), 1.0f);
glUniform1f(glGetUniformLocation(progID, "dmp_LightEnv.lutScaleD1"), 1.0f);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.clampHighlights"),
        GL_FALSE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightSource[0].geomFactor0"),
        GL_FALSE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightSource[0].geomFactor1"),
        GL_FALSE);

GLfloat LUT_RR[512], LUT_RG[512], LUT_RB[512], LUT_D1[512];
for( i = 0; i < 256; i++) LUT_RR[i] =
        z_schlick(0.7f, (float) i / 256.0f, true);
for( i = 0; i < 255; i++) LUT_RR[i + 256] = LUT_RR[i + 1] - LUT_RR[i];
LUT_RR[511] = z_schlick(0.7f, 1.0f, true) - LUT_RR[255];

for( i = 0; i < 256; i++) LUT_RG[i] =
        z_schlick(0.7f, (float) i / 256.0f, true);
for( i = 0; i < 255; i++) LUT_RG[i + 256] = LUT_RG[i + 1] - LUT_RG[i];
LUT_RG[511] = z_schlick(0.7f, 1.0f, true) - LUT_RG[255];

for( i = 0; i < 256; i++) LUT_RB[i] =
        z_schlick(0.7f, (float) i / 256.0f, true);
for( i = 0; i < 255; i++) LUT_RB[i + 256] = LUT_RB[i + 1] - LUT_RB[i];
LUT_RB[511] = z_schlick(0.5f, 1.0f, true) - LUT_RB[255];

for( i = 0; i < 256; i++) LUT_D1[i] =
        a_schlick(0.0015f, (float) i / 256.0f, true);
for( i = 0; i < 255; i++) LUT_D1[i + 256] = LUT_D1[i + 1] - LUT_D1[i];
LUT_D1[511] = a_schlick(0.0015f, 1.0f, true) - LUT_D1[255];

glBindTexture(GL_LUT_TEXTURE0_DMP, lutTexRR);
glTexImage1D(GL_LUT_TEXTURE0_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
        GL_LUMINANCEF_DMP, GL_FLOAT, LUT_RR);
glBindTexture(GL_LUT_TEXTURE1_DMP, lutTexRG);
glTexImage1D(GL_LUT_TEXTURE1_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
        GL_LUMINANCEF_DMP, GL_FLOAT, LUT_RG);
glBindTexture(GL_LUT_TEXTURE2_DMP, lutTexRB);
glTexImage1D(GL_LUT_TEXTURE2_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
        GL_LUMINANCEF_DMP, GL_FLOAT, LUT_RB);
glBindTexture(GL_LUT_TEXTURE3_DMP, lutTexD1);
glTexImage1D(GL_LUT_TEXTURE3_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
        GL_LUMINANCEF_DMP, GL_FLOAT, LUT_D1);
glUniform1i(glGetUniformLocation(progID,
        "dmp_FragmentMaterial.samplerRR"), 0);
glUniform1i(glGetUniformLocation(progID,
        "dmp_FragmentMaterial.samplerRG"), 1);
glUniform1i(glGetUniformLocation(progID,
        "dmp_FragmentMaterial.samplerRB"), 2);
glUniform1i(glGetUniformLocation(progID,
        "dmp_FragmentMaterial.samplerD1"), 3);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.bumpMode"),
        GL_LIGHT_ENV_BUMP_AS_BUMP_DMP);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.bumpSelector"),
        GL_TEXTURE0);
```

The `z_schlick` and `a_schlick()` functions are both Schlick functions.

This sample includes bump mapping because it needs perturbed tangents for input.

For more information about the Schlick anisotropic model, see the following reference.

Schlick, Christophe. An Inexpensive BRDF Model for Physically Based Rendering, Computer Graphics Forum, 13(3), pp. 233-246 (1994).

## 5.5. Subsurface-Scattering Model

When light enters an object made of a translucent material like skin, wax, or marble, the light is scattered internally in many directions before exiting the object. In such objects, light does not reflect from the same position at which it hits the object, so reflection cannot be represented using any of the models introduced so far.

This model, based on the light scattering theory proposed by Jensen et al., takes the integral of the incident light and the diffuse reflection function over a certain region of the surface. The region over which to integrate is of roughly the same size as the mean free path. For typical materials, this is between one and several millimeters.

When objects that are translucent, such as a human hand, are illuminated by a single light source against a dark background, the convex portions appear the most translucent. This is also true when using multiple light sources, so we can say that the most important part of rendering objects that have subsurface scattering is rendering their convex surfaces well.

With that in mind, if we consider the convex surfaces to be partial surfaces of spheres with radius S, we can take the integral of those surfaces using the formula proposed by Jensen et al. (radius S is much longer than the length of the "mean free path"). If we additionally consider the case of a rough surface, the reflection model can be expressed using the equation below.

$$Subsurface\_scattering = Rrgb(L \cdot N) \times T(V \cdot N)$$

$Rrgb(L \cdot N)$ is the sum of the Lambertian (diffuse light) term and the wrapping term. (The wrapping term represents penetration of light to surface areas where light is not directly incident.)

$$Rrgb(L \cdot N) = r(L \cdot N) + W(L \cdot N)$$

$$r = 0.5\alpha'\left(e^{-\beta} + e^{-\beta'}\right)$$

$$\beta = \sqrt{3(1-\alpha')}$$

$$\beta' = \beta B$$

$$B = \left(1 + \frac{4}{3}\frac{(1+Fdr)}{(1-Fdr)}\right)$$

$$W(L \cdot N) = \frac{\alpha' Y i_0}{4\pi(1 + i_1 h)}$$

$$Y = l\frac{\sqrt{1-(L \cdot N)^2}}{S}$$

$$h = \frac{L \cdot N}{Y}$$

$$i_0 = \sqrt{2\pi}\left(e^{-\beta}\beta^{-0.5} + Be^{-\beta'}\beta'^{-0.5}\right)$$

$$i_1 = \frac{\pi\left(e^{-\beta} + e^{-\beta'}\right)}{i_0}$$

See the references below for *α'* (albedo) and *Fdr*.

Jensen, H. W., Marschner, S., Levoy, M., and Hanrahan, P., A practical model for subsurface light transport, SIGGRAPH 2001 Proceedings, E. Fiume, Ed., Annual Conference Series, pp. 511–518.

**Code 5-6. Subsurface Scattering Model**

```
 /* Lighting Environment */
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.config" ),
    GL_LIGHT_ENV_LAYER_CONFIG7_DMP);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutEnabledRefl" ),
    GL_TRUE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.fresnelSelector" ),
    GL_LIGHT_ENV_PRI_SEC_ALPHA_FRESNEL_DMP);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.clampHighlights" ),
    GL_FALSE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutEnabledD0" ),
    GL_TRUE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutEnabledD1" ),
    GL_TRUE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.absLutInputRR" ),
    GL_FALSE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.absLutInputRG" ),
    GL_FALSE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.absLutInputRB" ),
    GL_FALSE);
/* below we use GL_TRUE to have nonzero for negative NV because negative NV values
happen on silhouette */
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.absLutInputD1" ),
    GL_TRUE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.absLutInputD0" ),
    GL_FALSE);
/* below we use GL_TRUE to have nonzero for negative NV because negative NV values
happen on silhouette */
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.absLutInputFR" ),
    GL_TRUE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutInputRR" ),
    GL_LIGHT_ENV_LN_DMP);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutInputRG" ),
    GL_LIGHT_ENV_LN_DMP);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutInputRB" ),
    GL_LIGHT_ENV_LN_DMP);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutInputD1" ),
    GL_LIGHT_ENV_NV_DMP);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutInputD0" ),
    GL_LIGHT_ENV_NH_DMP);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutInputFR" ),
    GL_LIGHT_ENV_NV_DMP);

/* Material */
GLfloat ms2[] = {0.28f, 0.28f, 0.28f, 1.f};
glUniform4fv(glGetUniformLocation(progID, "dmp_FragmentMaterial.specular1"), 1,
ms2);
glUniform1i(glGetUniformLocation(progID, "dmp_FragmentMaterial.samplerRR"), 0);
glUniform1i(glGetUniformLocation(progID, "dmp_FragmentMaterial.samplerRG"), 1);
glUniform1i(glGetUniformLocation(progID, "dmp_FragmentMaterial.samplerRB"), 2);
glUniform1i(glGetUniformLocation(progID, "dmp_FragmentMaterial.samplerD1"), 3);
glUniform1i(glGetUniformLocation(progID, "dmp_FragmentMaterial.samplerD0"), 4);
glUniform1i(glGetUniformLocation(progID, "dmp_FragmentMaterial.samplerFR"), 5);

/* Light Source */
GLfloat ld0[] = {1.f, 1.f, 1.f, 1.f}; /* light0 diffuse */
GLfloat ls0[] = {0.35f, 0.35f, 0.35f, 1.f}; /* light0 specular */
GLfloat ls1[] = {0.28f, 0.28f, 0.28f, 1.f}; /* light0 specular2 */
glUniform4fv(glGetUniformLocation(progID, "dmp_FragmentLightSource[0].diffuse"),
    1, ld0);
glUniform4fv(glGetUniformLocation(progID,
    "dmp_FragmentLightSource[0].specular0"), 1, ls0);
glUniform4fv(glGetUniformLocation(progID,
    "dmp_FragmentLightSource[0].specular1"), 1, ls1);
glUniform1i(glGetUniformLocation(progID,
    "dmp_FragmentLightSource[0].geomFactor0" ), GL_FALSE);
glUniform1i(glGetUniformLocation(progID,
    "dmp_FragmentLightSource[0].geomFactor1" ), GL_FALSE);

/* Texture Combiner 0 */
```

```
glUniform1i(glGetUniformLocation(progID, "dmp_TexEnv[0].combineRgb"),
    GL_ADD);
glUniform1i(glGetUniformLocation(progID, "dmp_TexEnv[0].combineAlpha"),
    GL_REPLACE);
glUniform3i(glGetUniformLocation(progID, "dmp_TexEnv[0].operandRgb"),
    GL_SRC_COLOR, GL_SRC_COLOR, GL_SRC_COLOR);
glUniform3i(glGetUniformLocation(progID, "dmp_TexEnv[0].operandAlpha"),
    GL_SRC_ALPHA, GL_SRC_ALPHA, GL_SRC_ALPHA);
glUniform3i(glGetUniformLocation(progID, "dmp_TexEnv[0].srcRgb"),
    GL_FRAGMENT_PRIMARY_COLOR_DMP, GL_FRAGMENT_SECONDARY_COLOR_DMP,
    GL_CONSTANT);
glUniform3i(glGetUniformLocation(progID, "dmp_TexEnv[0].srcAlpha"),
    GL_CONSTANT, GL_CONSTANT, GL_CONSTANT);

/* Texture Combiner 1 */
glUniform1i(glGetUniformLocation(progID, "dmp_Texture[0].samplerType" ),
    GL_TEXTURE_CUBE_MAP);
glUniform1i(glGetUniformLocation(progID, "dmp_TexEnv[1].combineRgb"),
    GL_MULT_ADD_DMP);
glUniform1i(glGetUniformLocation(progID, "dmp_TexEnv[1].combineAlpha"),
    GL_REPLACE);
glUniform3i(glGetUniformLocation(progID, "dmp_TexEnv[1].operandRgb"),
    GL_SRC_COLOR, GL_SRC_ALPHA, GL_SRC_COLOR);
glUniform3i(glGetUniformLocation(progID, "dmp_TexEnv[1].operandAlpha"),
    GL_SRC_ALPHA, GL_SRC_ALPHA, GL_SRC_ALPHA);
glUniform3i(glGetUniformLocation(progID, "dmp_TexEnv[1].srcRgb"),
    GL_TEXTURE0, GL_FRAGMENT_PRIMARY_COLOR_DMP, GL_PREVIOUS);
glUniform3i(glGetUniformLocation(progID, "dmp_TexEnv[1].srcAlpha"),
    GL_PREVIOUS, GL_PREVIOUS, GL_PREVIOUS);

/* LUT */
GLfloat qlut[3][512], lut[512];
int j, co;

GLuint lutids[6];
glGenTextures(6, lutids);
/* RR, RG, RB */
for (co = 0; co < 3; co++) {
    for (j = 0; j < 128; j++) {
        LN = (float) j/128.f;
        qlut[co][j] = calc_lamb(LN, co) + calc_wrap(LN, co);
    }
    for (j = 128; j < 256; j++) {
        LN = (float) (j - 256) /128.f;
        qlut[co][j] = calc_wrap(LN, co);
    }
    for (j = 0; j < 127; j++)
        qlut[co][j + 256] = qlut[co][j + 1] - qlut[co][j];
    qlut[co][127 + 256] = calc_lamb(1.f, co) + calc_wrap(1.f, co) -
        qlut[co][127];
    for (j = 128; j < 255; j++)
        qlut[co][j + 256] = qlut[co][j + 1] - qlut[co][j];
    qlut[co][255 + 256] = qlut[co][0] - qlut[co][255];
}
glBindTexture(GL_LUT_TEXTURE0_DMP, lutids[0]);
glTexImage1D(GL_LUT_TEXTURE0_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
    GL_LUMINANCEF_DMP, GL_FLOAT, qlut[0]);
glBindTexture(GL_LUT_TEXTURE1_DMP, lutids[1]);
glTexImage1D(GL_LUT_TEXTURE1_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
    GL_LUMINANCEF_DMP, GL_FLOAT, qlut[1]);
glBindTexture(GL_LUT_TEXTURE2_DMP, lutids[2]);
glTexImage1D(GL_LUT_TEXTURE2_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
    GL_LUMINANCEF_DMP, GL_FLOAT, qlut[2]);
/* D1 */
for (j = 0; j < 256; j++) lut[j] = calc_t((float) j / 255.9375f);
for (j = 0; j < 255; j++) lut[j + 256] = lut[j + 1] - lut[j];
lut[255 + 256] = calc_t(1.f) - lut[255];
glBindTexture(GL_LUT_TEXTURE3_DMP, lutids[3]);
glTexImage1D(GL_LUT_TEXTURE3_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
    GL_LUMINANCEF_DMP, GL_FLOAT, lut);
/* D0 */
memset(lut, 0, sizeof(lut));
for (j = 0; j < 128; j++) lut[j] = beckmann((float)j / 128.f, 0.5f);
for (j = 128; j < 256; j++) lut[j] = 0.f;
for (j = 0; j < 127; j++) lut[j + 256] = lut[j + 1] - lut[j];
lut[127 + 256] = 1.f - lut[127];
for (j = 128; j < 256; j++) lut[j + 256] = 0;
glBindTexture(GL_LUT_TEXTURE4_DMP, lutids[4]);
glTexImage1D(GL_LUT_TEXTURE4_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
    GL_LUMINANCEF_DMP, GL_FLOAT, lut);
```

```
/* FR */
memset(lut, 0, sizeof(lut));
for (j = 0; j < 256; j++)
lut[j] = r_fresnel((float)j / 255.9375f, 2.f, 0.35f, 0.f);
for (j = 0; j < 255; j++) lut[j + 256] = lut[j + 1] - lut[j];
    lut[255 + 256] = r_fresnel(1.f, 2.f, 0.35f, 0.f) - lut[255];
glBindTexture(GL_LUT_TEXTURE5_DMP, lutids[5]);
glTexImage1D(GL_LUT_TEXTURE5_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
    GL_LUMINANCEF_DMP, GL_FLOAT, lut);
```

The `calc_lamb`, `calc_wrap`, and `calc_t()` functions are the functions used to calculate r(L · N), W(L · N), and T(L · N) respectively.

The reflection model is represented by the RR, RG, RB, and D1 lookup tables. D0 represents the effect of specular light and FR represents the effect of peripheral light.

Combiner `0` sums the object's primary color with the secondary color calculated by the reflection model. Specify `GL_MULT_ADD_DMP` for the color operand in combiner 1 to calculate and combine the surrounding lighting effects with the object's color.

## 5.6. Toon Shading

Toon shading is a way of shading where only two or three colors are used, creating blocks of color instead of smooth gradations. To apply toon shading, we use a function that outputs set values corresponding to specific resulting ranges of the dot products N · H or L · N. We use this step function to generate a lookup table whose outputs are stepped values. By multiplying the lookup table outputs by constant colors, we can shade a scene with clearly defined flat colors, much like a cartoon.

If changes in shading are caused solely by changes in the position of the lights, use the dot product L · N (`GL_LIGHT_ENV_LN_DMP`). If changes in the line of sight are also a factor, use the dot product N · H (`GL_LIGHT_ENV_NH_DMP`).

If you only plan to change the brightness of the specular 1 value of the lights, set the stepped values in the reflection (RR) lookup table and simply specify a layer configuration where each component of the reflection uses that lookup table. Alternately, using separate lookup tables for each component makes it possible to express extreme changes in coloring, even when the color step function is the same in each table.

You can render the outlines of a toon-shaded object using only specular light by rendering areas darkly (in black) where the orientations of the view vector and the normal differ greatly. Do this by taking the dot product N · V (`GL_LIGHT_ENV_NV_DMP`) as the input and setting up the distribution factor lookup tables (D0, D1) to yield either `0.0` (for all inputs close to `0.0`) or `1.0` (for all other inputs).

**Code 5-7. Toon Shading**

```
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.config"),
        GL_LIGHT_ENV_LAYER_CONFIG2_DMP);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutEnabledRefl"),
        GL_TRUE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.absLutInputD0"),
        GL_TRUE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.absLutInputD1"),
        GL_TRUE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.absLutInputRR"),
        GL_FALSE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutInputRR"),
        GL_LIGHT_ENV_LN_DMP);
```

```
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutInputD0"),
        GL_LIGHT_ENV_LN_DMP);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.lutInputD1"),
        GL_LIGHT_ENV_NV_DMP);
glUniform1i(glGetUniformLocation(progID,
        "dmp_FragmentLightSource[0].geomFactor0"), GL_FALSE);
glUniform1i(glGetUniformLocation(progID,
        "dmp_FragmentLightSource[0].geomFactor1"), GL_FALSE);
glUniform1i(glGetUniformLocation(progID,
        "dmp_FragmentLightSource[0].spotEnabled"), GL_FALSE);
glUniform1i(glGetUniformLocation(progID, "dmp_LightEnv.clampHighlights"),
        GL_FALSE);

float delta[] = {1.0f, 0.7f, 0.5f, -1.0f};
for (i = 0, j = 127; j >= 0; j--) {
    LN = (float) j / 128.0f ;
    if (LN > delta[i]) lut[j] = previous;
    else {
        lut[j] = LN; previous = lut[j]; i++;
    }
}
for (j = 0 ; j < 127; j++) lut[j + 256] = lut[j+1] - lut[j];
lut[127 + 256] = 0.0f;
for (i = 0, j = 255; j >= 128; j--) {
    LN = (float)(j - 256) /128.0f;
    if (LN > delta[i]) lut[j] = previous;
    else {
        lut[j] = LN; previous = lut[j]; i++;
    }
}
for (j = 128; j < 255; j++) lut[j + 256] = lut[j+1] - lut[j];
lut[255 + 256] = lut[0] - lut[255];
glBindTexture(GL_LUT_TEXTURE0_DMP, lutids[0]);
glTexImage1D(GL_LUT_TEXTURE0_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
        GL_LUMINANCEF_DMP, GL_FLOAT, lut);
glUniform1i(glGetUniformLocation(progID, "dmp_FragmentMaterial.samplerRR"), 0);

float highlight_eps = 0.01f;
for (j = 0; j < 256; j++)
    if ((float) j / 256.0f <= 1.0f - highlight_eps) lut[j] = 0.0f;
    else                                            lut[j] = 1.0f;
for (j = 0; j < 255; j++) lut[j + 256] = lut[j+1] - lut[j];
lut[255 + 256] = 0.0f;
glBindTexture(GL_LUT_TEXTURE1_DMP, lutids[1]);
glTexImage1D(GL_LUT_TEXTURE1_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
        GL_LUMINANCEF_DMP, GL_FLOAT, lut);
glUniform1i(glGetUniformLocation(progID, "dmp_FragmentMaterial.samplerD0"), 1);

float outline = 0.15f;
for (j = 0; j < 256; j++)
    if ((float) j / 256.0f < outline) lut[j] = 0.0f;
    else                              lut[j] = 1.0f;
for (j = 0; j < 255; j++) lut[j + 256] = lut[j+1] - lut[j];
lut[255 + 256] = 1.0f - lut[255];
glBindTexture(GL_LUT_TEXTURE2_DMP, lutids[2]);
glTexImage1D(GL_LUT_TEXTURE2_DMP, 0, GL_LUMINANCEF_DMP, 512, 0,
        GL_LUMINANCEF_DMP, GL_FLOAT, lut);
glUniform1i(glGetUniformLocation(progID, "dmp_FragmentMaterial.samplerD1"), 2);
```

As shown in the sample, `delta` is the array of dot products that produce stepped results, `highlight_eps` is the range of dot products taken as highlights, and `outline` is the range of dot products taken as shadows. Use these values to change how toon shading is rendered.



# 6. Stereoscopic Display

Of the two LCD screens on the 3DS system, the upper screen can display images that appear in stereoscopic 3D to the naked eye, with no need for any special equipment.

Two images must be rendered for stereoscopic display, one for the left eye and one for the right. The CTR-SDK includes the ULCD library, which takes the camera matrix created for normal display and from it calculates the camera matrices needed to render these two images. Always use the camera matrices calculated by the ULCD library when rendering. This additionally serves to standardize the method used for stereoscopic display among various applications.

This chapter describes the principles of stereoscopic display, how to implement stereoscopic display in an application, and how to coordinate your application with the stereo cameras.

For convenience, the terms defined below are used in regard to stereoscopic display.

#### Real Space

Stereoscopic display involves various real physical factors, such as the distance between the player and the surface of the upper screen. "Real space" refers to the space in the real world where the player and 3DS system exist, distinct from the space created within an application.

#### Virtual Space

In contrast to real space, "virtual space" refers to the space created within an application.

#### Base Camera

This refers to the camera configured or created by the application for a scene. The ULCD library calculates the camera matrices for the left and right eyes based on information from this base camera.

#### Base Plane

During stereoscopic display, the plane in virtual space that represents the 3DS system's LCD screen is called the "base plane." The base plane is a cross-section of the camera's viewing volume.

#### 3D Depth Slider

This is a slider switch on the 3DS system for adjusting the intensity of the stereoscopic effect. This slider allows different people to adjust the stereoscopic display for the optimal view, or to adjust the effect if feeling any fatigue after extended play. The ULCD library uses the input value from the slider to adjust the distance between the left and right cameras generated in virtual space.

#### Optimal Viewing Position

This refers to the viewing position from which the player can best perceive the effect of stereoscopic display.

## 6.1. Principles of Stereoscopic Display

Stereoscopic display is the rendering of an object while factoring in the difference in view between the left and right eyes (this difference in view is called "parallax") to generate the illusion of distance

between the object and the viewpoint. The ULCD library offers two different ways of calculating the camera matrices used in stereoscopic rendering: one that keeps the base camera settings unchanged as much as possible (application priority method), and another that automatically changes the base camera settings as needed (realism priority method).

This section describes the principles underlying each of these calculation methods.

## 6.1.1. Preconditions

The ULCD library assumes the following conditions when calculating the camera matrices used for stereoscopic display.

- The distance $Dist_{eye}$ between the player's eyes is 62 mm.
- Let $Dist_{e2d}$ be the distance between the center of the surface of the LCD screen and the player's eyes.
- $Depth_{ltd}$ is the limit distance at which depth is naturally perceivable by human eyes, according to existing research. Likewise, $P_{r\_ltd}$ is the limit parallax in real space (on the base plane) that is required to produce this depth. **The CTR Guidelines define the maximum parallax values that are allowed, both for the effect of distance away from the player (depth into the screen), and the effect of distance toward the player (jumping out of the screen).**
- Let $Len_{disp}$ be the length of the shorter side of the upper screen.

## 6.1.2. Distance Between the Left and Right Cameras and Calculating Each Camera's Viewing Volume

The following information is needed as input to calculate the camera matrices.

- The view matrix $View_{base}$ from which the parallax images for stereoscopic display are generated.
- The projection matrix $Proj_{base}$ used as the base from which the left-eye and right-eye camera viewing volumes are generated.
- The distance $D_{level}$ in virtual space from the camera to a point that you want to position on the surface of the LCD.
- A coefficient $D_r$ for adjusting the level of stereoscopy (its value ranges from `0.0` through `1.0`).

You can find the width and height of the base plane from $D_{level}$ and the viewing volume parameters $l_{base}$, $r_{base}$, $b_{base}$, $t_{base}$, $n_{base}$, and $f_{base}$ (left, right, bottom, top, near, far) that can be reverse calculated from the projection matrix $Proj_{base}$.

Width of the base plane:

$$W_{level} = | r_{base} - l_{base} | \times D_{level} / n_{base}$$

Height of the base plane:

$$H_{level} = | t_{base} - b_{base} | \times D_{level} / n_{base}$$

From this height and width, and from the actual dimensions of the upper screen, we can find the coefficient for converting the real space scale to the virtual space scale.

$$Scale_{r2v} = H_{level} / Len_{disp}$$

## 6.1.2.1. Application Priority Calculation Method

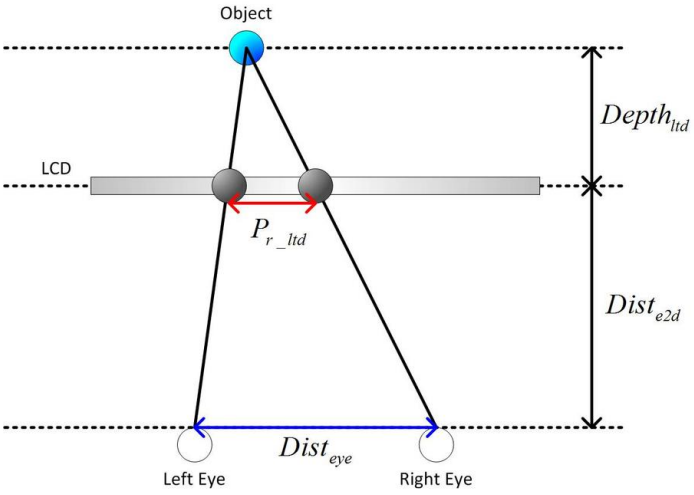
This calculation method prioritizes maintaining the view from the base camera (the view originally expected by the application) when it generates the camera matrices for rendering the images for the left and right eyes.

The limit parallax in real space (the greatest parallax that still feels natural) $P_{r\_ltd}$ can be calculated from the preconditions.

$$P_{r\_ltd} = Dist_{eye} \times ( Depth_{ltd} / ( Dist_{e2d} + Depth_{ltd} ) )$$

Figure 6-3 shows the positions and relationships between the terms of these equations.

**Figure 6-1. Limit Parallax in Real Space**




You can use $Scale_{r2v}$ to convert $P_{r\_ltd}$ to the virtual-space limit parallax $P_{v\_ltd}$.

$$P_{v\_ltd} = P_{r\_ltd} \times Scale_{r2v}$$

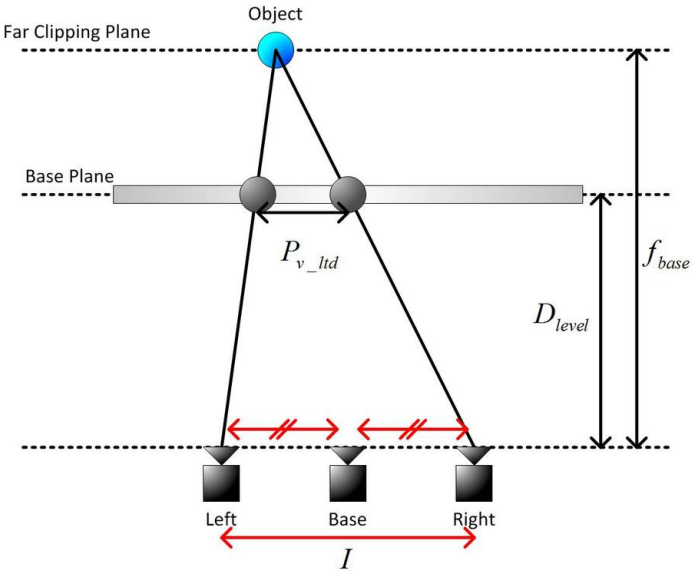
Calculate the distance $I$ between the left and right cameras such that the base-plane parallax for displaying objects on the far clipping plane is set to this limit parallax, without changing the depth position of the base camera. Use $D_{level}$ in this calculation. If the far clipping plane is in an abnormal position (for example closer than the base plane) this distance will be 0.

The equation to find $I$ is as follows:

$$I = P_{v\_ltd} \times ( f_{base} / ( f_{base} - D_{level} ) )$$

The following figure shows these relationships.

**Figure 6-2. Distance Between the Left and Right Cameras in Virtual Space**




In this method, parameters related to the viewing volume do not change because the position of the base camera is not changed.

Near clipping plane width: $W_n = | r_{base} - l_{base} |$

Near clipping plane height: $H_n = | t_{base} - b_{base} |$

## 6.1.2.2. Realism Priority Calculation Method

This method generates camera matrices that render the left and right images so that objects on the base plane appear to look the same as they would in real space. In other words, with this method the cameras are adjusted for the most natural stereoscopic view, assuming that the player is viewing at a distance $Dist_{e2d}$ from the CTR LCD.

The distance between the player's two eyes and the distance from their eyes to the CTR LCD in real space are reflected in the relationships between the left and right cameras and base plane in virtual space. The parameters set for the base camera are automatically changed as part of this process.

You can calculate $D_{level\_new}$ by converting the distance $Dist_{e2d}$ between the CTR LCD surface and the player's eyes to the scale of the virtual space.

$D_{level\_new} = Dist_{e2d} \times Scale_{r2v}$

Use this calculated distance, $D_{level}$, and $Proj_{base}$ to find the distances to the clipping planes. In turn, use these to find distances $n_{new}$ and $f_{new}$ to the clipping planes of the newly generated camera.

$n_{new} = D_{level\_new} - ( D_{level} - n_{base} )$ , $f_{new} = D_{level\_new} + ( f_{base} - D_{level} )$

If the newly calculated near clipping plane is behind the cameras, adjust the clipping planes as follows.

$n_{new} = D_{level\_new} \times 0.01$

Likewise, adjust as follows if the far clipping plane is closer than the near clipping plane.

$f_{new} = n_{new} \times 2.0$

Because you are not moving the base plane, the base camera is moved forward or backward as

needed to satisfy the values obtained so far.

Next find the dimensions of the near clipping plane (width, height), which have changed because it was moved. Also recalculate the near clipping plane's range (left, right, top, bottom).

Near clipping plane width:

$$W_{n\_new} = W_{level} \times n_{new} / D_{level\_new}$$

Near clipping plane height:

$$H_{n\_new} = H_{level} \times n_{new} / D_{level\_new}$$

Near clipping plane range:

$$l_{new} = tmp \times l_{base} \ , \ r_{new} = tmp \times r_{base} \ ,$$

$$t_{new} = tmp \times t_{base} \ , \ b_{new} = tmp \times b_{base}$$

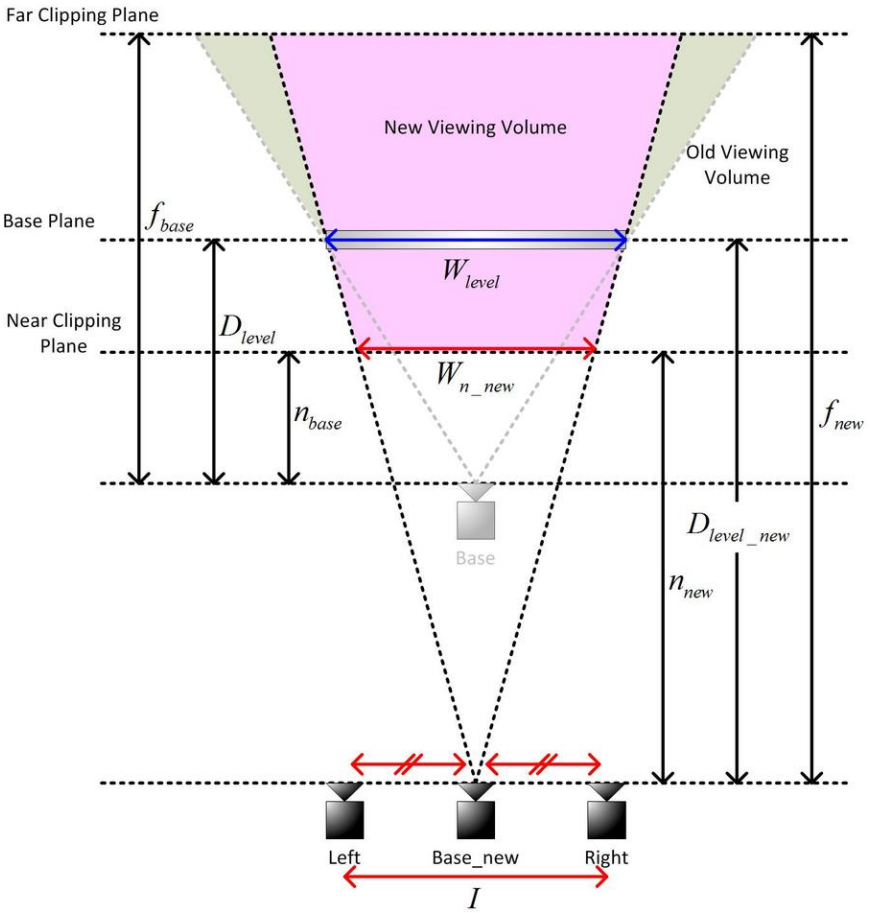
$$tmp = H_{n\_new} / \mid t_{base} - b_{base} \mid$$

The following equation applies the distance between the player's eyes to the distance $I$ between the left and right cameras in virtual space.

$$I = Dist_{eye} \times Scale_{r2v}$$

Figure 6-3 shows the positions and relationships between the terms of these equations.

**Figure 6-3. Realism Priority Calculation Method**




## 6.1.3. Generating Projection Matrices

The adjustment coefficient *vol* and the value of the 3DS system's 3D depth slider $D_r$ are applied to the distance *l* between the left and right cameras, which you found earlier. There is no sense of depth at the lowest 3D depth slider value, at which the right and left cameras match the base camera.

$$l = l \times vol \times D_r \quad (0.0 \leq vol \leq 1.0)$$

The parameters of the base camera's viewing volume differ depending on which calculation method is used, but for convenience, the following uniform notations are used: the near clipping plane width is $W_n$, the near clipping plane height is $H_n$, the distance from the base camera to the base plane is $D_{level}$, and the parameters related to the viewing volume are *l*, *r*, *t*, *b*, *n*, and *f* for the left, right, top, bottom, near, and far planes, respectively. To get the respective viewing volumes for the left and right cameras, find the parallax $P_n$ at the near clipping plane. Assume that the left and right cameras are equidistant from the base camera position.

$$P_n = l \times 0.5 \times (( D_{level} - n ) / D_{level} )$$

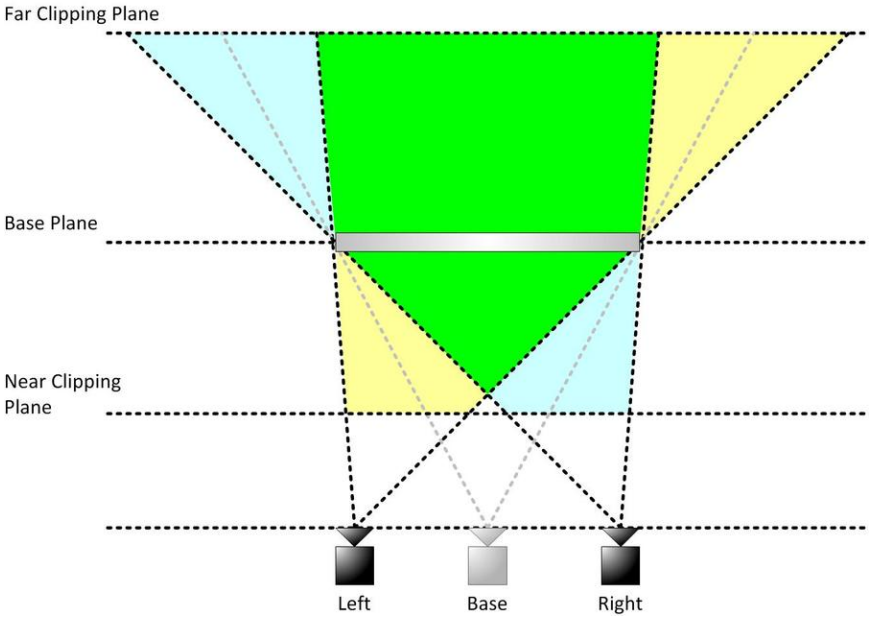
Based on this parallax, you can calculate the position of the near clipping plane in the respective left and right camera viewing volumes. The top, bottom, near, and far planes are common between the left and right cameras because these cameras have only been moved horizontally.

Position of the near clipping plane for the left camera: $l_{left} = ( l - P_n ) + l \times 0.5$, $r_{left} = ( r - P_n ) + l \times 0.5$

Position of the near clipping plane for the right camera: $l_{right} = ( l + P_n ) - l \times 0.5$, $r_{right} = ( r + P_n ) - l \times 0.5$

Calculate the projection matrices for the left and right cameras based on parameters that follow from these equations. The figure below shows the viewing volumes yielded thus far. To generate a stereoscopic image, an object must be placed in the region where the viewing volumes intersect (shown by the central portion of the figure).

**Figure 6-4. Viewing Volumes for the Left and Right Cameras**




## 6.1.4. Generating View Matrices

Generate the left and right camera view matrices using the distance *l* between the left and right cameras and the base camera information derived from $View_{base}$. This base camera information consists of its position $Pos_{base}$, its direction $Dir_{base}$, and its right-hand direction $E_{right}$, which are all 3D vectors. $Dir_{base}$ and $E_{right}$ are unit vectors with a length of 1.

When the distance between the cameras has been found using the realism priority method, the left and right camera positions may have moved forward or backward (along the depth direction) relative to the original base camera. The base camera's position is updated as follows.

$$Pos_{base} = Pos_{base} - ( D_{level\_new} - D_{level} ) \times Dir_{base}$$

Because the base camera is midway between the left and right cameras, the positions of the left and right cameras relative to the base camera will be equal to the base camera's position, plus or minus half of the distance between the left and right cameras. In the following equations, the left and right camera positions are $Pos_{left}$ and $Pos_{right}$, and the direction of their look-at points are $Tgt_{left}$ and $Tgt_{right}$, respectively (we do not need to find the precise position of the look-at points because we only need to know the camera orientation).

$$Pos_{left} = Pos_{base} - l \times 0.5 \times E_{right} \ , \ Tgt_{left} = Pos_{left} + Dir_{base}$$

$$Pos_{right} = Pos_{base} + l \times 0.5 \times E_{right} \ , \ Tgt_{right} = Pos_{right} + Dir_{base}$$

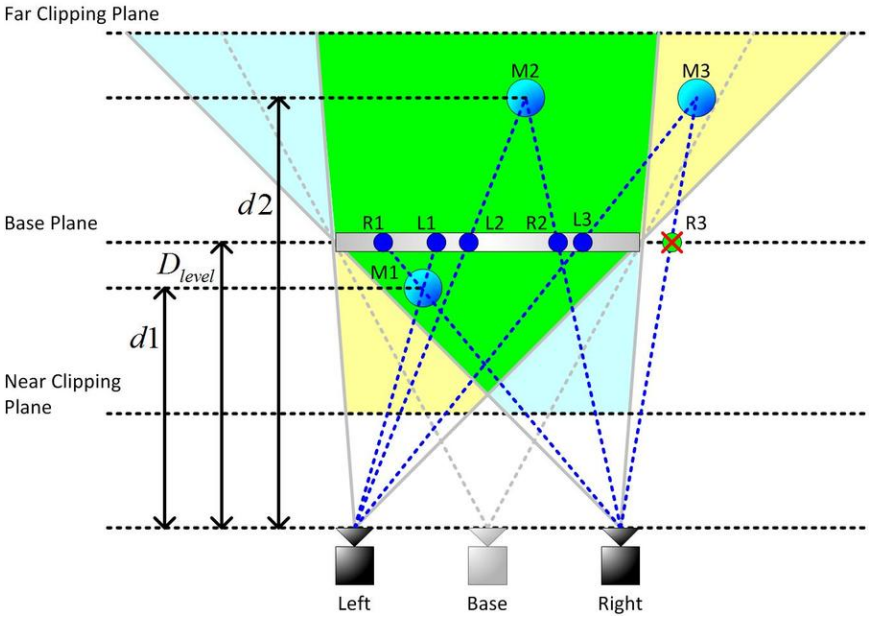
Using these equations, you can generate the view matrices for the left and right cameras.

## 6.1.5. Parallax Required to Display an Object at an Arbitrary Position

Using the matrices calculated by the ULCD library, you can implement stereoscopic display without keeping track of how much parallax is necessary for each 3D object rendered in perspective projection. However, to stereoscopically display a 2D image or a 3D object rendered using orthographic projection, your application must create the left-eye and right-eye images itself.

The figure below shows the parallax required to display objects "in front of" the LCD screen (closer to the viewer than the screen) or "behind" the LCD screen (farther from the viewer than the screen), using the left and right cameras generated by the library.

**Figure 6-5. Parallax Used to Display Objects at Arbitrary Positions**



Object M1 is positioned at distance *d*1 from the cameras. To display M1 closer to the viewer than the LCD is, render M1 at the positions where straight lines connecting the cameras to M1 would intersect the base plane. These positions are R1 and L1 for the right and left eyes, respectively. In other words, to display M1 in front of the LCD, the parallax R1L1 is required. You can calculate this parallax length by taking the distance *I* between the left and right cameras and the distance $D_{level}$ from the cameras to the base plane, and using trigonometry.

$$R1L1 = I \times ( D_{level} - d1 ) / d1 \text{ , but } D_{level} \geq d1$$

Similarly, you can use the parallax R2L2 to display object M2, which is positioned at distance *d*2 (behind the LCD).

$$R2L2 = I \times ( d2 - D_{level} ) / d2 \text{ , but } D_{level} < d2$$

If an object is positioned like M3 and included in only one of the left/right camera viewing volumes, only one camera can pass a line through it and intersect the base plane. It is not possible to implement stereoscopic parallax for such objects.

### 6.1.6. Appropriate Parallax for Objects Located at the Maximum Possible Distance From the Base Plane

This section explains the appropriate degree to offset the left-eye and right-eye images (whether of a 2D image or an object rendered using orthogonal projection) from each other to make that image or object appear to have the maximum possible stereoscopic depth, where it is located the maximum possible distance from the viewer.

Objects positioned at maximum possible depth can be represented by offsetting the left and right images by a distance equal to the limit parallax $P_{r\_ltd}$ (this distance is the actual physical offset on the surface of the LCD, which is also the base plane). In other words, to represent a far-off background with stereoscopic display enabled, the images rendered to the render buffers for the left and right eyes are each offset from the position of the base camera by half of the limit parallax. In practice, however, you must use $P_{v\_ltd}$ (the result of converting $P_{r\_ltd}$ to the scale of virtual space) as the offset during rendering.

In principle, this technique can also be used to display stereoscopic images that appear to be jumping out in front of the LCD screen (toward the user) to the farthest extent possible. However, you must be careful when doing so because objects that appear to be located in front of the screen do not have as wide a display range as those that appear to be located behind the screen. This difference in range is shown in Figure 6-5.

## 6.2. How to Implement in an Application

You must follow the general process below to implement stereoscopic display in your application.

- Set the display mode of the upper screen to stereoscopic display.
- Prepare a display buffer for displaying the right-eye image.
- For perspective projection, use the ULCD library to calculate the camera matrices for the left and right eyes.

- Render the images for the left and right eyes, and transfer them to the corresponding display buffers.
- Display the content of the display buffers on the LCD.

## 6.2.1. Setting the Display Mode

Call the nngxSetDisplayMode() function to set the display mode and enable or disable stereoscopic display on the upper screen. If you change the mode during display, do so just after a V-Sync operation.

**Code 6-1. Setting the Display Mode**

```
void nngxSetDisplayMode(GLenum mode);
```

Specify NN_GX_DISPLAYMODE_STEREO for the *mode* parameter to enable stereoscopic display. Specify NN_GX_DISPLAYMODE_NORMAL to disable stereoscopic display. Specifying any other value results in a GL_ERROR_9003_DMP error. The default mode is NN_GX_DISPLAYMODE_NORMAL, in which stereoscopic display is disabled.

Enabling stereoscopic display causes two screens—one for the left eye and one for the right eye—to both be displayed on the upper LCD screen. The resolutions, formats, and memory region locations (whether in main memory, VRAM-A, or VRAM-B) must be the same for both of these screens. If they are not the same, a call to the nngxSwapBuffers() function results in an error. Note that this has no effect on the lower screen.

To specify the screen for the left eye, use NN_GX_DISPLAY0, which is the same value that indicates normal (non-stereoscopic) display. Use the added value NN_GX_DISPLAY0_EXT to specify the screen for the right eye. When you specify the screen for the right eye, you must still perform the same sequence of operations that are required for other screens. This sequence includes specifying the screen (nngxActiveDisplay), binding the display buffer (nngxBindDisplayBuffer), and setting the offset for LCD output (nngxDisplayEnv), among other required processing.

To make it easier to tell which screen you are specifying, the SDK defines aliases: NN_GX_DISPLAY0_LEFT is equivalent to NN_GX_DISPLAY0, and NN_GX_DISPLAY0_RIGHT is equivalent to NN_GX_DISPLAY0_EXT.

## 6.2.2. Allocating Display Buffers

When stereoscopic display is enabled, the upper LCD screen displays different images for the left and right eyes. Applications handle the image that is seen by the left eye and the image that is seen by the right eye as images displayed on separate LCD screens. For this reason, specifying NN_GX_DISPLAY0 in calls to functions now specifies the left-eye screen, instead of simply specifying the upper screen as this value did formerly. To specify the right-eye screen, use the new value NN_GX_DISPLAY0_EXT.

Because the left-eye and right-eye screens are actually treated as separate LCD screens, they need separate display buffers. When using multibuffering, note that the number of required display

buffers will increase accordingly.

The following code sample shows how to allocate a display buffer of the size of the upper LCD screen and use it for double-buffering the right-eye screen.

**Code 6-2. Allocating the Display Buffer for the Right Eye**

```
GLuint m_Display0BuffersExt[2];
// For right eye - Upper (DISPLAY0_EXT)
nngxActiveDisplay(NN_GX_DISPLAY0_EXT);
nngxGenDisplaybuffers(2, m_Display0BuffersExt);
nngxBindDisplaybuffer(m_Display0BuffersExt[0]);
nngxDisplaybufferStorage(GL_RGB8_OES,
    NN_GX_DISPLAY0_WIDTH, NN_GX_DISPLAY0_HEIGHT, NN_GX_MEM_FCRAM);
nngxBindDisplaybuffer(m_Display0BuffersExt[1]);
nngxDisplaybufferStorage(GL_RGB8_OES,
    NN_GX_DISPLAY0_WIDTH, NN_GX_DISPLAY0_HEIGHT, NN_GX_MEM_FCRAM);
nngxDisplayEnv(0, 0);
```

## 6.2.3. Using the ULCD Library

When the application is rendering in perspective projection, it can use the matrices calculated by the ULCD library to implement stereoscopic display. Because the ULCD library uses the base camera to calculate the left and right cameras and associated information needed for stereoscopic display, the application does not need to keep track of how much parallax is necessary.

The ULCD library provides the `nn::ulcd::StereoCamera` class, which creates camera matrices that account for parallax. You must include the `nn/ulcd.h` header file to use this class in your application. Note that a function to get the 3D depth slider value was made public as of CTR-SDK 3.3.1, but it is intended for adding secondary effects to 3D viewing. For this reason, unless you use this class, you cannot use the 3D depth slider to adjust the intensity of the stereoscopic effect in your application.

**Note:** If you intend to use the function that gets the 3D depth slider value, you must contact Nintendo in advance.

### 6.2.3.1. Initializing

Generate an instance of the `nn:ulcd::StereoCamera` class and call the `Initialize` member function to initialize.

**Code 6-3. Initialization**

```
void Initialize(void);
```

This function initializes all the internally stored information.

### 6.2.3.2. Setting and Getting the Limit Parallax

Use the member functions `SetLimitParallax` and `GetLimitParallax` to set and get the limit parallax.

**Code 6-4. Setting and Getting the Limit Parallax**

```
void SetLimitParallax(const f32 limit);
f32  GetLimitParallax(void) const;
```

For the *limit* parameter, specify the limit parallax to set, in millimeters. The maximum limit parallax is set forth in the CTR Guidelines, and for the depth direction (into the screen) you can set any positive value up to this maximum. The limit parallax set here affects the calculated camera matrices.

The `GetLimitParallax()` function gets the current limit parallax value. If the `SetLimitParallax()` function has not yet been called, the function returns the maximum value (for depth into the screen) that is set forth in the guidelines.

## 6.2.3.3. Base Camera Information

Use the `SetBaseFrustum()` and `SetBaseCamera()` member functions to set information for the base camera.

**Code 6-5. Setting the Base Camera**

```
void SetBaseFrustum(const nn::math::Matrix44 *proj);
void SetBaseFrustum(const f32 left, const f32 right, const f32 bottom,
                    const f32 top, const f32 near, const f32 far);
void SetBaseCamera(const nn::math::Matrix34 *view);
void SetBaseCamera(const nn::math::Vector3 *position,
                   const nn::math::Vector3 *rightDir,
                   const nn::math::Vector3 *upDir,
                   const nn::math::Vector3 *targetDir);
```

The `SetBaseFrustum()` function mainly sets the near and far clipping planes. You can either set each of the parameters individually, or pass in a projection matrix created by the `nn::math::MTX44Frustum` or `nn::math::MTX44Perspective()` functions, from which the parameters will be calculated. If you specify a projection matrix in `nn::math::Matrix44` format, the base frustum is not calculated correctly unless the projection matrix was calculated based on the definition of the viewing volume of the 3DS graphics system. (Specifically, the z-coordinate must be clipped to the range from `0` to `–Wc`.)

The `SetBaseCamera()` function mainly sets the position and other parameters of the base camera. You can either set each of the parameters individually, or pass in a view matrix created by a function such as `nn::math::MTX34LookAt`, from which the parameters will be calculated. When specifying a view matrix or specifying vectors, make sure that you use the right-handed coordinate system adopted by the 3DS graphics system.

## 6.2.3.4. Calculating the Left and Right Camera Matrices

As described in 6.1 Principles of Stereoscopic Display, the SDK provides two methods for calculating the left and right camera matrices: one that keeps the base camera settings unchanged as much as possible (application priority method), and another that automatically

changes the base camera settings (realism priority method).

Use the `CalculateMatrices` member function to calculate matrices that prioritize the application, and use the `CalculateMatricesReal` member function to calculate matrices that prioritize realism.

**Code 6-6. Calculating the Left and Right Camera Matrices**

```
void CalculateMatrices(
        nn::math::Matrix44* projL, nn::math::Matrix34* viewL,
        nn::math::Matrix44* projR, nn::math::Matrix34* viewR,
        const f32 depthLevel, const f32 factor,
        const nn::math::PivotDirection pivot = nn::math::PIVOT_UPSIDE_TO_TOP,
        const bool update3DVolume = true);
void CalculateMatricesReal(
        nn::math::Matrix44* projL, nn::math::Matrix34* viewL,
        nn::math::Matrix44* projR, nn::math::Matrix34* viewR,
        const f32 depthLevel, const f32 factor,
        const nn::math::PivotDirection pivot = nn::math::PIVOT_UPSIDE_TO_TOP,
        const bool update3DVolume = true);
```

Both functions take the same arguments and differ only in their results.

For the *projL* and *viewL* parameters, specify the storage location of the projection matrix and view matrix for the left eye. For the *projR* and *viewR* parameters, specify the corresponding matrices for the right eye. The functions write values to these matrices, so you need instances for these structures.

Set *depthLevel* equal to the distance $D_{level}$ from the base camera to the base plane, and set *factor* equal to the stereoscopic adjustment coefficient $D_r$. The factor parameter is used to correct internally calculated results. If this parameter is set equal to `0.0`, there is zero parallax. If this parameter is set equal to `1.0`, there is zero correction of the results. In addition to the effects of this value, the calculated results are also affected by the input value from the 3D depth slider.

These functions multiply the output projection matrix by a rotation matrix so that the camera's upward direction matches the direction specified in the *pivot* parameter. The default argument value is `nn::math::PIVOT_UPSIDE_TO_TOP`, which creates a rotation matrix in which the camera's upward direction matches the upper screen's upward direction after the projection matrix is rotated. (The upper screen's upward direction is the direction from its center toward the center of its long side, opposite the lower screen.) Specify `nn::math::PIVOT_NONE` if rotation is unnecessary, such as when rotation has been factored into the base camera settings.

The *update3DVolume* parameter specifies whether to get and use the 3D depth slider value when calculating the matrix. The value is used (`true`) by default if no argument is specified. When the 3D depth slider value is used for matrix calculations, if your implementation calls these functions multiple times per frame, any changes to the 3D depth slider value in the middle of a calculation could affect the rendering result.

For implementations that use this technique, disable (specify `false`) the *update3DVolume* parameter and call the `Update3DVolume` member function at the start of each frame. If you do not call this function, adjustments to the 3D depth slider will not affect the rendering result.

**Code 6-7. Updating the 3D Depth Slider Value**

```
void Update3DVolume(void);
```

## 6.2.3.5. Getting Parallax Information

The class provides member functions that use the results from the most recent matrix calculation to get the parallax required to produce an image at the specified distance from the camera. These functions cannot get the parallax if the class has since been initialized with the `Initialize()` function.

**Code 6-8. Getting Parallax Information**

```
f32 GetParallax(const f32 distance) const;
f32 GetMaxParallax(void) const;
```

The `GetParallax()` function gets the parallax required to place an object at a position separated from the camera by *distance*, and returns the ratio of this parallax to the LCD screen width. The `GetMaxParallax()` function returns this ratio for an object at maximum possible depth. In the return value, `1.0` is equivalent to 100 percent of the screen width. Multiply the return value by the LCD screen resolution to find how many pixels to shift the object to the left and the right from its position, as seen by the base camera, to render it stereoscopically.

The return value is positive if the position specified by *`distance`* is behind the base plane, and negative if the position is in front of the base plane. The return value is `0` if *`distance`* is negative.

When rendering a 3D object in perspective projection, developers previously did not need to worry about parallax when using the matrices generated by the library, but when rendering 2D objects or when rendering in an orthographic projection, you will have to calculate the parallax for each object. However, calling the `GetParallax()` function for each object to calculate the parallax causes a resource intensive CPU load. To speed up this process, use the function below to obtain a partially calculated parallax value, and shift the remainder of the calculation to the vertex shader.

**Code 6-9. Getting a Partially Calculated Parallax Value**

```
f32 GetCoefficientForParallax(void) const;
```

Multiply the return value by ((*distance* – *depthLevel*) ÷ *distance*) to derive the same value returned by the `GetParallax()` function.

Like the functions that get parallax, there are member functions that get the distance from the base camera to the base plane, near clipping plane, and far clipping plane. All these distances are based on the most recent calculation results.

**Code 6-10. Getting the Distance From the Base Camera to the Base Plane, Near Clipping Plane, and Far Clipping Plane**

```
f32 GetDistanceToLevel(void) const;
f32 GetDistanceToNearClip(void) const;
f32 GetDistanceToFarClip(void) const;
```

These functions are mainly used to get the updated base camera information when using the realism priority method. When using the application priority method, these functions simply return the information originally passed to the calculation.

### 6.2.3.6. Finalization

After an instance of the `nn::ulcd::StereoCamera` class is no longer needed, call the `Finalize` member function.

**Code 6-11. Finalization**

```
void Finalize(void);
```

Always call this function explicitly from the application, even though it is also called by the destructor.

## 6.2.4. Rendering and Transferring to Display Buffers

When you render 3D objects using perspective projection, you can use the projection and view matrices calculated for the left and right cameras by the ULCD library. This allows you to render without needing to keep track of the parallax. However, to render a 2D object or to render a 3D object using orthographic projection, your application must keep track of the parallax because it must render the left-eye and right-eye images itself. 2D objects can be rendered without needing to know parallax, if you render them using perspective projection, but you must keep track of the positions of objects so you can take into account their foreground/background relationships.

Images are rendered to color buffers and transferred to display buffers for display on the LCD screen. Note that the images for the left and right eyes must be transferred to different display buffers.

## 6.2.5. Displaying on the LCD Screen

Bind the display buffers for the left and right eyes to the screens specified by `NN_GX_DISPLAY0` and `NN_GX_DISPLAY0_EXT`, respectively. Next you must swap the display buffers, but because during stereoscopic display the right-eye display buffer is handled simultaneously, when the upper screen (`NN_GX_DISPLAY0`) is specified, you do not need to also pass `NN_GX_DISPLAY0_EXT` to the `nngxSwapBuffers()` function.

**Code 6-12. Binding Display Buffers and Swapping Buffers**

```
// UpperLCD for Left Eye
nngxActiveDisplay(NN_GX_DISPLAY0);
nngxBindDisplaybuffer(m_Display0Buffers[m_CurrentDispBuf0]);
// UpperLCD for Right Eye
nngxActiveDisplay(NN_GX_DISPLAY0_EXT);
nngxBindDisplaybuffer(m_Display0BufferExt[m_CurrentDispBuf0Ext]);
// Swap buffers
nngxSwapBuffers(NN_GX_DISPLAY0);
```

## 6.2.6. 3D Depth Slider Input

Tune your application so that players who have maximized the 3D depth slider (and who have no difficulty with stereoscopic display at this setting) enjoy your recommended "best" level of the effect.

When the 3D depth slider is used to reduce the 3D depth until the screen switches from stereoscopic (3D) display to normal 2D display, the system automatically turns off the LCD shutter. The system also automatically handles switching from 2D to 3D display and changing the brightness of the LCD backlight, so you do not need to control these from the application, other than for debugging purposes.

## 6.2.7. Disabling Stereoscopic Display

When the display of 3D images is restricted by Parental Controls in System Settings, and also when the 3D depth slider has been moved all the way down to 0 (the minimum value), stereoscopic display is forcibly disabled. In this state, attempts to set the display mode or to display the right-eye image on the LCD screen are ignored, and normal (2D) display is used without exception.

You can call the `nn::gx::IsStereoVisionAllowed()` function to determine whether stereoscopic display is enabled. This function returns a value of `true` when stereoscopic display is enabled, and `false` when stereoscopic display has been forcibly disabled. This status is not affected by the display mode setting.

## 6.2.8. Flipping Left-to-Right

You cannot simply invert the x-axis of the projection matrix to flip a scene left-to-right, and then stereoscopically display it. This causes positions in the depth direction to be inverted about the base plane.

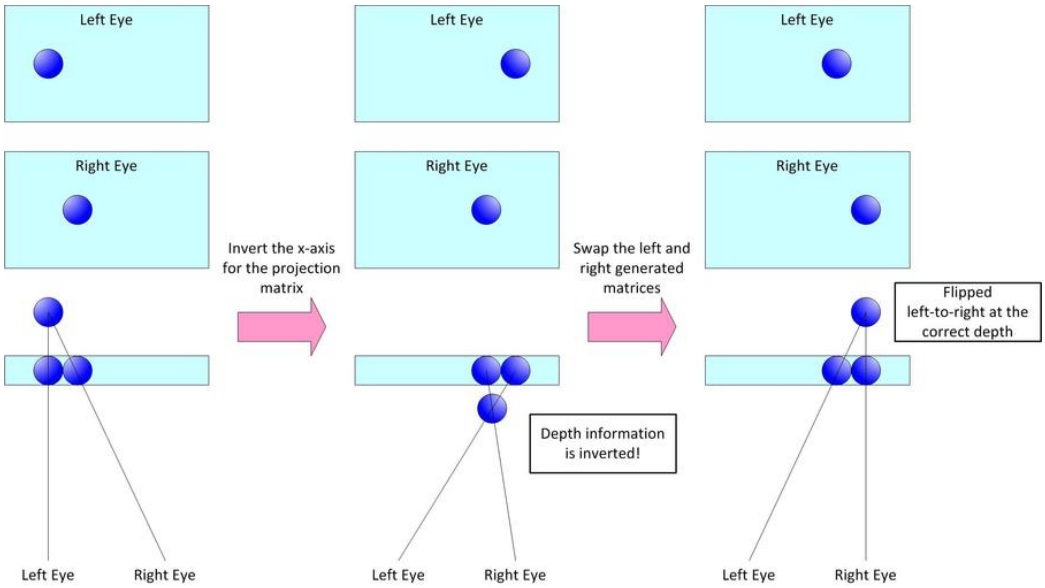
To resolve this problem, use a projection matrix with an inverted s-axis to create modelview and projection matrices for stereoscopic display, and then switch those left and right matrices.

**Code 6-13. Flipping Left-to-Right in Stereoscopic Display**

```
nn::math::Matrix44 proj,rev;
nn::math::Matrix44 projL, projR;
nn::math::Matrix34 viewL, viewR;
// SetBaseFrustum
MTX44Frustum(&proj, l, r, b, t, n, f);
MTX44Identity(&rev);
rev.m[0][0] = -1.0f;
MTX44Mult(&proj, &proj, &rev);          // Flip on the x-axis.
s_StereoCamera.SetBaseFrustum(&proj);
// SetBaseCamera
nn::math::Matrix34 cam;
nn::math::Vector3 camUp(0.f, 1.f, 0.f);
nn::math::MTX34LookAt(&cam, &camPos, &camUp, &focus);
s_StereoCamera.SetBaseCamera(&cam);
// CalculateMatrices
s_StereoCamera.CalculateMatrices(
    &projR, &viewR, &projL, &viewL,     // Specify left/right matrices in reverse
order.
    depthLevel, factor, nn::math::PIVOT_UPSIDE_TO_TOP);
```

The following figure shows the principles of flipping left-to-right in stereoscopic display.

**Figure 6-6. Principles of Flipping Left-to-Right in Stereoscopic Display**


Left Eye
Right Eye
Invert the x-axis for the projection matrix
Swap the left and right generated matrices
Flipped left-to-right at the correct depth
Depth information is inverted!
Left Eye
Right Eye

## 6.3. Using the Reserved Fragment Shader

When the rendering pipeline uses shadows, the shadow textures consist of depth values in relation to lights in the scene, so the changes to the view matrices caused by stereoscopic display have no effect on shadow texture generation. View matrices are pertinent to object rendering, so you must render twice, once for the left eye and once for the right. When doing so, you must be careful with the texture coordinate transformation matrices that you use to apply the shadow textures.

When the rendering pipeline uses gas rendering, the view matrices are involved in the generation of depth values during the first pass, so to use stereoscopic display, all passes must be run separately for the left and right eyes.

When the rendering pipeline uses fog, the lookup table input values are depth values in window coordinates, so you must regenerate the lookup table whenever the projection matrix is updated because of stereoscopic display. If you are using the application priority method, you can create the lookup table from the base camera's projection matrix, even in stereoscopic display. However, if you are using the realism priority method, the near and far clipping planes change from the base camera values, so you must regenerate the lookup table, even when the only change is an adjustment to the 3D depth slider.

For fragment lighting in general, if light positions are affected by the view matrices, the light positions you use when you render must take into account both the left and right view matrices. (This issue is also affected by how you standardize coordinate systems.)

## 6.4. Coordinating With the Stereo Cameras

The left and right cameras are connected to two different ports. Note that port 1 captures images for the right eye (`NN_GX_DISPLAY0_EXT`), and port 2 for the left eye (`NN_GX_DISPLAY0`). The cameras output image data to their respective buffers from these two ports, but because there is only one YUVtoRGB circuit, the application must take measures such as mutual exclusion when converting

YUV-format images captured by the cameras into RGB format. The left and right images must also be obtained from the same frame, or stereoscopic display may not work properly.

Simply displaying the images obtained from the left and right cameras as is also produces a stereoscopic display, but note that this produces a different distance between the two cameras as compared to the distance between the user's eyes.

## 6.4.1. Calibrating the Stereo Cameras

The two outer cameras are designed to be positioned horizontally 35 mm apart, but manufacturing mounting tolerances may produce some degree of deviation, leading to a mismatch between the left and right camera images. The application must correct for this mismatch because the camera library does not automatically correct for such mounting variance.

> **Note:** Perfect correction might not be possible, leaving some mismatch around the edges of corrected images.

Call the `nn::camera::GetStereoCameraCalibrationData()` function to get the stereo camera calibration data. The data is stored in an `nn::camera::StereoCameraCalibrationData` structure.

**Code 6-14. Getting the Stereo Camera Calibration Data**

```
void nn::camera::GetStereoCameraCalibrationData(
        nn::camera::StereoCameraCalibrationData * pDst);
```

Taken together, the calibration data member variables indicate the correction required to align the left and right camera images (zoom, optical axis rotation, translation) and the measurement conditions for the correction values.

**Table 6-2. Bit Arrays and Post-Concatenation Values in Individual Mode**

| Item | Member | Range of Obtainable Valid Values |
|---|---|---|
| Zoom/shrink | `Scale` | 0.9604 through 1.0405 |
| Z-axis (optical axis) rotation angle | `rotationZ` | –1.6 through +1.6 (degrees) |
| Amount of horizontal (x-axis) translation | `translationX` | –154 through –19 (pixels) |
| Amount of vertical (y-axis) translation | `TranslationY` | –70 through +70 (pixels) |

The range of possible values for two vertical translations is quite broad, so just moving the image from the left camera and adjusting could lead to problems such as the image edges being visible within the display area, which results in rendering incomplete edges of the display. When using calibration data for 3D display, use the CAMERA library functions for calculating correction matrices. These include a function to expand an image to the display size if the edges are incomplete, even when the stereo camera placement error is at the limit. When using calibration data for purposes other than 3D display (such as for image recognition), be careful to avoid possible bugs or errors, such as reduced processing accuracy resulting from certain calibration data values.

Call the following functions to use the calibration data to calculate the correction matrix that must

be multiplied against the left camera image.

**Code 6-15. Calculating the Correction Matrix**

```
void nn::camera::GetStereoCameraCalibrationMatrix(
        nn::math::MTX34 * pDst,
        const nn::camera::StereoCameraCalibrationData & cal,
        const f32 translationUnit, const bool isIncludeParallax = true);
void nn::camera::GetStereoCameraCalibrationMatrixEx(
        nn::math::MTX34 * pDstR, nn::math::MTX34 * pDstL, f32 * pDstScale,
        const nn::camera::StereoCameraCalibrationData & cal,
        const f32 translationUnit, const f32 parallax,
        const s16 orgWidth, const s16 orgHeight,
        const s16 dstWidth, const s16 dstHeight);
f32 nn::camera::GetParallax(
        const nn::camera::StereoCameraCalibrationData & cal, f32 distance);
```

You can get the correction matrix in the *pDst* parameter by calling the `nn::camera::GetStereoCameraCalibrationMatrix()` function with the calibration data passed to the *cal* parameter and the amount of translation in 3D space (see Note) necessary to move by one pixel passed to the *translationUnit* parameter. Pass `true` in the *isIncludeParallax* parameter to include the parallax from the measurement chart in the correction matrix. Including this parallax places the camera focus 250 mm away from the cameras. Pass `false` in the *isIncludeParallax* parameter to only correct for mounting mismatch.

**Note:** The *translationUnit* parameter specifies the value of the amount of translation (the amount of translation required in 3D space to move the camera image by one pixel) multiplied by the ratio between the VGA image and display image widths. However, this is the method for specifying the value when displaying an image as is at a different size than the VGA image, or when zooming an image after applying a correction matrix. If you are instead zooming an object with an image applied before the image has had a correction matrix applied, and you want the subject to be displayed in the image at the same size as the subject displayed pixel-by-pixel in the VGA image, specify the amount of translation without modification.

The `nn::camera::GetStereoCameraCalibrationMatrixEx()` function gets a correction matrix that allows stereoscopic images to be displayed at the edge of the screen when the camera placement error has reached its limit. Because this function uses the parallax obtained by the `nn::camera::GetParallax()` function, the correction matrix is calculated to place the focus at a particular distance. We recommend that you use this function to calculate the correction matrix, unless you have some reason to do otherwise.

The `nn::camera::GetStereoCameraCalibrationMatrixEx()` function takes the same values in *cal* and *translationUnit* as `nn::camera::GetStereoCameraCalibrationMatrix`. Use the `GetParallax()` function to calculate the parallax (in pixels) of the VGA images to pass to the *parallax* parameter. The parallax is calculated with the focal point at the distance (in meters) specified by the *distance* parameter. Specify the width and height (in pixels) of the (trimmed) camera image in *orgWidth* and *orgHeight*, respectively. Specify the necessary width and height (in pixels) for rendering in *dstWidth* and *dstHeight*.

The correction matrices to be multiplied with the left and right cameras are returned in *pDstL* and *dDstR*, respectively. The scaling factor necessary for the rendering size is returned in *pDstScale*.

**Note:** For more information about how the correction matrix is calculated, see the *CTR-SDK API Reference*.

Call the `nn::camera::GetParallaxOnChart()` function to get the parallax from the measurement chart (the parallax that causes the cameras to focus on objects 250 mm away) in pixels.

**Code 6-16. Getting Parallax From the Measurement Chart**

```
f32 nn::camera::GetParallaxOnChart(
        const nn::camera::StereoCameraCalibrationData & cal);
```

The horizontal mounting variance is equal to the result of subtracting the `translationX` member of the calibration data from the value obtained by this function.

## 6.4.2. Linking Stereo Camera Brightness

Automatic exposure and other internal processing for the left camera is independent of the same processing for the right camera, and consequently the brightness differs between the left and right cameras for some subjects. The CAMERA library provides a function, `nn::camera::SetBrightnessSynchronization`, to automatically link the brightness of stereo camera images.

**Code 6-17. Linking Stereo Camera Brightness**

```
nn::Result nn::camera::SetBrightnessSynchronization(bool enable);
```

When *`enable`* is `true`, the two cameras use the same brightness (they are linked). The default setting is disabled (`false`). This is disabled (`false`) by default, and only works for the stereo cameras.

You can call this function even when the (left and right) outer cameras have not been started. After the two cameras have been linked, this setting is remembered, even if the cameras later enter standby mode. They will continue to be linked when they are restarted.

This feature links the brightness of stereo camera images by periodically writing the exposure setting for the right outer camera to the left outer camera. In particular, the exposure for the right outer camera changes the exposure for the left outer camera when auto-exposure is enabled. The exposure settings may be delayed by heavy processing because they are linked by a low-priority thread created by the library, which does not consume application resources.

This function call fails for any of the following camera settings. After the cameras have been linked, each of the following settings will fail.

- White-balance setting other than `WHITE_BALANCE_NORMAL`
- Disabled auto-white balance
- Capture mode of `PHOTO_MODE_LANDSCAPE`
- Contrast setting of `CONTRAST_PATTERN_10`

**Warning:** Processing may block in the library for a long time if this function is called while the cameras are being restarted.

# 6.5. Notes

**Note:** The content of this section may be revised or expanded to comply with guidelines that are being reviewed separately.

## 6.5.1. Placing Objects in Front of the LCD Screen

When you use stereoscopic display to position objects so that they appear to jump out in front of the surface of the LCD, you must position them so that they do not touch any edge of the LCD screen. If an object does overlap the edge of the screen, the screen's border, which appears (from the viewer's point of view) to be behind the object, covers up an object that is supposed to be in front of it. This causes an unnatural effect, destroying the illusion of 3D depth for the player.

You must also be careful when positioning the near clipping plane. Normally the near clipping plane is very close to the camera. However, this usual distance to the near clipping plane does not match the boundary distance from the screen toward the player that allows the player to experience the correct 3D sense of depth. When an object gets too close to the camera, stereoscopic display of the object becomes impossible.

This is not a problem that can be solved by simply placing the near clipping plane far from the camera. In each individual application, you must consider and apply ways to prevent objects from getting too close to the camera.

## 6.5.2. Positioning When 2D Objects and 3D Objects Are Displayed Together

It is possible to display 2D and 3D objects at the same time by combining orthographic projection with objects rendered at camera settings that vary in a way that mimics perspective projection. When you do this, you must adjust the position, size, and order with which you render the 2D objects to ensure that the 2D and 3D objects have the proper foreground/background relationship.

If you use perspective projection to render the 2D objects in addition to the 3D objects, you do not have to adjust the 2D objects themselves. However, you must still consider the foreground/background relationship of the 3D and 2D objects when deciding where to position your 2D objects. This is because 2D objects could be unintentionally hidden by 3D objects.

## 6.5.3. Handling a Tilted System

Although the system can be tilted toward or away from the user without any problems while stereoscopic images are being displayed, it may become difficult to display stereoscopic images if the system is tilted left and right or rotated because the LCD will no longer face the user directly.

To handle this problem, Face Raiders (installed on the system) uses the gyro sensor to determine whether the system has been tilted. If so, it is designed to make the 3D adjustment coefficients weak (decrease the strength of the stereoscopic effect). We recommend that you use the gyro

sensor to detect how the system is tilted and adjust the stereoscopic strength accordingly if your application displays stereoscopic images while the system is being moved aggressively.

### 6.5.4. Cautions When Using the Display Mode to Disable 3D Display

The `nn::ulcd::CalculateMatrices()` function returns values based on the 3D volume, even when the application calls `nngxSetDisplayMode(NN_GX_DISPLAYMODE_NORMAL)` to disable 3D display. Consequently, using the matrices calculated by the ULCD library (as is) for display results in on-screen changes linked to 3D volume, even though the 3D LED is turned off.

You can avoid this situation by either not using `CalculateMatrices` after disabling 3D display, by calling `nngxSetDisplayMode`, or by passing in a value of `0.0` for the *factor* parameter.

This issue does not occur when 3D display is forcibly disabled, as described in 6.2.7 Disabling Stereoscopic Display.



# 7. Description of the ETC1 Compressed Texture Format

ETC1 compressed textures are compressed using the Ericsson Texture Compression (ETC) algorithm. This format is well suited to hardware decompression, allowing for higher-speed processing than conventional textures. This algorithm can compress each 4x4 texel block of a 24-bit color image (RGB8) down to just 8 bytes (64 bits), helping reduce texture memory requirements. However, the compression method applied by this algorithm assumes that the colors within each individual block tend to be similar, making it more suited to compression of photographs of landscapes or people than to images that have very different RGB component values (such as both red and blue areas) within the same blocks.

That said, this compression method works well for the 3DS system's own image format that adds a 4-bit alpha value to each texel (for 128 bits in total).

This chapter describes the ETC1 compressed texture format and the format conversions needed for use in the 3DS system.

## 7.1. ETC Format

The ETC format splits each 4x4 texel block into 4x2 or 2x4 sub-blocks. For each sub-block it stores a table containing the sub-block's base color (an RGB value) and delta values as compared to that base color. The format has two modes, individual mode and differential mode, but the only difference is in the format of the base color contained in the first 32 bits. The latter 32 bits are the same in both modes, and the decompression algorithm is also the same, aside from how it decodes the base color.

The following figure shows the ETC1 compressed texture bit layout. The data is arranged with the alpha channel coming before the color channel.

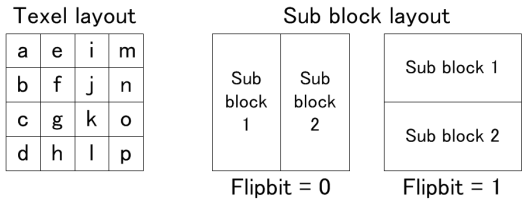
**Figure 7-1. Bit Layout in ETC1 Compressed Textures**

## 7.1.1. Texel Layout and Sub-Blocks

The latter 32 bits are further subdivided into MSB (most significant bit) and LSB (least significant bit) regions, of 16 bits each. In each region, the upper-left texel corresponds to the least significant bit. Bits increase in significance as you move from top to bottom down the texels, and the lower-right texel corresponds to the most significant bit. In other words, the least significant bits in each region (bits 0 and 16) correspond to the texel in the upper left, with the next texel down corresponding to the next-most-significant bit. Upon reaching the bottom of this column of texels, the next texel is at the top of the next column to the right, until reaching the most significant bits (bits 15 and 31) corresponding to the texel in the lower right of the block.

Depending on the flipbit (bit 32) value, the blocks are subdivided either vertically (2x4) or horizontally (4x2). These groups of eight subdivided texels are called *sub-blocks*, with either the left or top sub-block designated as sub-block 1, and the right or bottom sub-block as sub-block 2. The texel layout order differs depending on which way the block was subdivided.

**Figure 7-2. Texel and sub-block Layout**




## 7.1.2. Determining the Base Color

The base color is determined separately for each sub-block, and the method for determining the format and base color depends on the diffbit value (bit 33).

With a diffbit value of 0, the format is individual mode and the R, G, and B components of the base color for each sub-block are assigned 4 bits each. Sub-block 1 is (R1, G1, B1) and sub-block 2 is (R2, G2, B2). The base color is the 8-bit value that results from concatenating this 4-bit value twice. (This binary operation is equivalent to multiplying the decimal value of the original 4 bits by 17.) With an R1 value of 1001b (9), the red component of the base color for sub-block 1 would be 10011001b (153).

With a diffbit value of 1 the format is differential mode. In this mode, the R, G, and B components for sub-block 1 are assigned 5 bits each. For sub-block 2, 3 bits are assigned to each component, with the values being the difference (delta values) as compared to the 5-bit values for sub-block 1. For sub-block 1 values (R1', G1', B1'), the sub-block 2 values are (R1'+dR2, G1'+dG2, B1'+dB2). The sub-block 1 base color is determined by concatenating the most-significant 3 bits of each component value after the initial 5 bits, for a total of 8 bits (effectively multiplying the original 5-bit value by 8.25). The sub-block 2 base color is determined by adding the 3-bit delta, expressed as a two's complement value (between +3 and -4), to the 5 bits of the sub-block 1 value, and then concatenating the most-significant 3 bits of each component value after the initial 5 bits. So for an R1' value of 11001b (25) and a dR2 value of 100b (-4), the base color red component of sub-block 1 is 11001110b (206), and the base color red component of sub-block 2 is 11001b (25) - 100b (4) = 10101b (21), which then becomes 10101101b (173). Combinations where the result of adding the delta value falls outside the range from 0 through 31 cause unstable operations. Such combinations must be avoided during compression.

**Table 7-1. Bit Arrays and Post-Concatenation Values in Individual Mode**

| Bit Arrays | Values | Bit Arrays | Values | Bit Arrays | Values | Bit Arrays | Values |
|---|---|---|---|---|---|---|---|
| 0000b | 0 | 0100b | 68 | 1000b | 136 | 1100b | 204 |
| 0001b | 17 | 0101b | 85 | 1001b | 153 | 1101b | 221 |
| 0010b | 34 | 0110b | 102 | 1010b | 170 | 1110b | 238 |
| 0011b | 51 | 0111b | 119 | 1011b | 187 | 1111b | 255 |

**Table 7-2. Bit Arrays, Post-Concatenation Values, and Values+Delta in Differential Mode**

| Bit Arrays | Values | 100b (–4) | 101b (–3) | 110b (–2) | 111b (–1) | 000b (0) | 001b (+1) | 010b (+2) | 011b (+3) |
|---|---|---|---|---|---|---|---|---|---|
| 00000b | 0 | - | - | - | - | 0 | 8 | 16 | 24 |
| 00001b | 8 | - | - | - | 0 | 8 | 16 | 24 | 33 |
| 00010b | 16 | - | - | 0 | 8 | 16 | 24 | 33 | 41 |
| 00011b | 24 | - | 0 | 8 | 16 | 24 | 33 | 41 | 49 |
| 00100b | 33 | 0 | 8 | 16 | 24 | 33 | 41 | 49 | 57 |
| 00101b | 41 | 8 | 16 | 24 | 33 | 41 | 49 | 57 | 66 |
| 00110b | 49 | 16 | 24 | 33 | 41 | 49 | 57 | 66 | 74 |
| 00111b | 57 | 24 | 33 | 41 | 49 | 57 | 66 | 74 | 82 |
| 01000b | 66 | 33 | 41 | 49 | 57 | 66 | 74 | 82 | 90 |
| 01001b | 74 | 41 | 49 | 57 | 66 | 74 | 82 | 90 | 99 |
| 01010b | 82 | 49 | 57 | 66 | 74 | 82 | 90 | 99 | 107 |
| 01011b | 90 | 57 | 66 | 74 | 82 | 90 | 99 | 107 | 115 |
| 01100b | 99 | 66 | 74 | 82 | 90 | 99 | 107 | 115 | 123 |
| 01101b | 107 | 74 | 82 | 90 | 99 | 107 | 115 | 123 | 132 |
| 01110b | 115 | 82 | 90 | 99 | 107 | 115 | 123 | 132 | 140 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 01111b | 123 | 90 | 99 | 107 | 115 | 123 | 132 | 140 | 148 |
| 10000b | 132 | 99 | 107 | 115 | 123 | 132 | 140 | 148 | 156 |
| 10001b | 140 | 107 | 115 | 123 | 132 | 140 | 148 | 156 | 165 |
| 10010b | 148 | 115 | 123 | 132 | 140 | 148 | 156 | 165 | 173 |
| 10011b | 156 | 123 | 132 | 140 | 148 | 156 | 165 | 173 | 181 |
| 10100b | 165 | 132 | 140 | 148 | 156 | 165 | 173 | 181 | 189 |
| 10101b | 173 | 140 | 148 | 156 | 165 | 173 | 181 | 189 | 198 |
| 10110b | 181 | 148 | 156 | 165 | 173 | 181 | 189 | 198 | 206 |
| 10111b | 189 | 156 | 165 | 173 | 181 | 189 | 198 | 206 | 214 |
| 11000b | 198 | 165 | 173 | 181 | 189 | 198 | 206 | 214 | 222 |
| 11001b | 206 | 173 | 181 | 189 | 198 | 206 | 214 | 222 | 231 |
| 11010b | 214 | 181 | 189 | 198 | 206 | 214 | 222 | 231 | 239 |
| 11011b | 222 | 189 | 198 | 206 | 214 | 222 | 231 | 239 | 247 |
| 11100b | 231 | 198 | 206 | 214 | 222 | 231 | 239 | 247 | 255 |
| 11101b | 239 | 206 | 214 | 222 | 231 | 239 | 247 | 255 | - |
| 11110b | 247 | 214 | 222 | 231 | 239 | 247 | 255 | - | - |
| 11111b | 255 | 222 | 231 | 239 | 247 | 255 | - | - | - |

## 7.1.3. Determining the Delta Tables and Texel Colors

The final texel colors are determined by adding delta values selected from the delta table to the base color.

The delta table is specified by a "table codeword" (TableCW) section for each sub-block. Bits 37 through 39 are used for TableCW1 for sub-block 1, and bits 34 through 36 are used for TableCW2 for sub-block 2, with each 3-bit region allowing for 8 different bit values. Four delta values are given by the delta table. The delta values for each texel are determined using these in combination with the MSB and LSB. The selected delta value is added to all the RGB components of the base color, and the resulting values are clamped to the 0 through 255 range. For example, given a delta table value of 011b, an MSB of 1, and an LSB of 1, -42 is added to all components of the base color. Assuming a base color of (33, 198, 99), the resulting texel color is (0, 156, 57).

**Table 7-3. TableCW, MSB, and LSB Combinations and Delta Values**

| TableCW | MSB=1 LSB=1 | MSB=1 LSB=0 | MSB=0 LSB=0 | MSB=0 LSB=1 |
|---|---|---|---|---|
| 000b | -8 | -2 | +2 | +8 |
| 001b | -17 | -5 | +5 | +17 |
| 010b | -29 | -9 | +9 | +29 |
| 011b | -42 | -13 | +13 | +42 |
| 100b | -60 | -18 | +18 | +60 |
| 101b | -80 | -24 | +24 | +80 |
| 110b | -106 | -33 | +33 | +106 |
| 111b | -183 | -47 | +47 | +183 |

# 7.2. PICA Native Format Compressed Textures

When using ETC1 compressed textures on the 3DS system, the format must be PICA native format. PICA native format follows the OpenGL ES standard specification, except for differences in the lookup origin and block layout for texels, and in the byte order.

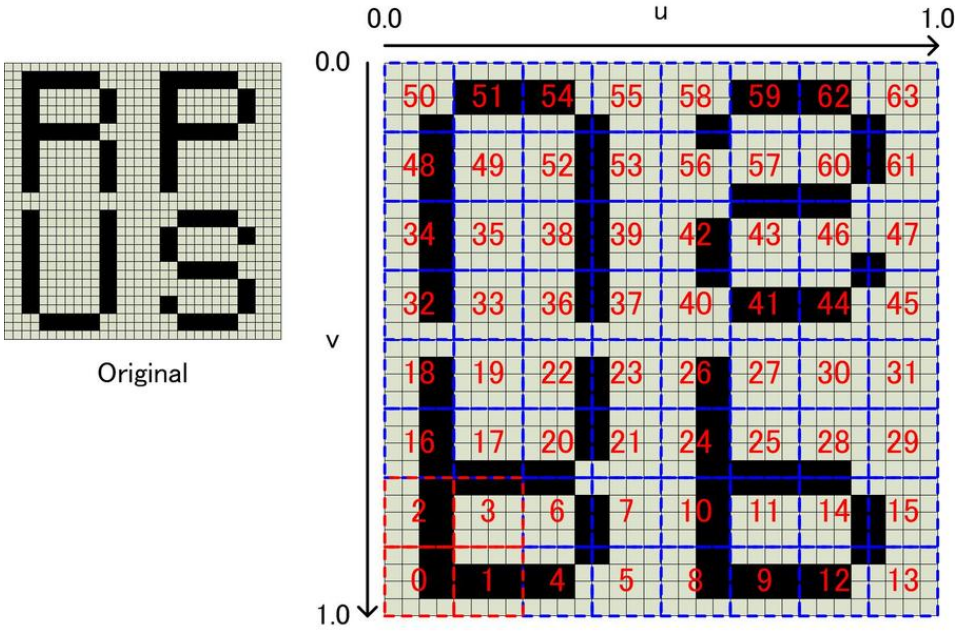
## 7.2.1. V-Flipping

The texel lookup origin in the OpenGL ES specification is (u, v) = (0.0, 0.0), but on the 3DS system, this is (u, v) = (0.0, 1.0). You must first flip a texture image in the v coordinate direction (V-flip) before compressing it.

## 7.2.2. Block Format

ETC1-compressed textures define a metablock as two 4x4 texel blocks each in the u and v directions, for a total of 8x8 texels. Blocks within a metablock are laid out in a zigzag pattern, with the metablocks themselves laid out consecutively in the u direction.

**Figure 7-3. Block Lookup Order in ETC1 Compressed Textures**




## 7.2.3. Byte Order

The 3DS byte order differs from the OpenGL ES specification, due to differing endianness. Looking at the bit layout in terms of the actual byte order reveals the following points.

**Figure 7-4. Bit Layout In Terms of Actual Byte Order**

The color channel data is byte-swapped in units of 8 bytes, but alpha channel data is not byte-swapped. Alpha channel data still comes before color channel data.



# 8. Command Caches

Command caches allow you to reuse the 3D commands that have accumulated in a command list, while rendering 3D graphics. A cache reuses the 3D commands themselves, so caches allow you to reduce the cost of the function calls required to generate 3D commands, and thereby reduce CPU load.

Command caches may be used for the following.

- Saving Command Lists
- Reusing Command Lists
- Copying Command Lists
- Exporting and importing command lists.
- Adding 3D Commands
- Editing 3D Commands

This chapter introduces how to use command caches, and provides the information needed to edit 3D commands.

## 8.1. Saving Command Lists

Command caches cache both the contents of the 3D Command Buffer stored in a command list and

the queued Command Requests. However, there is no such thing as a command cache object. Command caches simply use `start saving` and `stop saving` declarations to get the information that is required to reuse a command list. Because a command cache simply reuses the data accumulated in its targeted command list without modifying that data, clearing or destroying that command list causes the command cache to stop functioning (unless the application has copied the data).

Use the `nngxStartCmdlistSave()` function to start saving a command list.

**Code 8-1. Start Saving a Command List**

```
void nngxStartCmdlistSave(void);
```

The `GL_ERROR_8034_DMP` error is issued if the `nngxStartCmdlistSave()` function is called while a command list is already in the process of being saved. The `GL_ERROR_8035_DMP` error is issued if a command list is bound that has an object name of `0`.

Use the `nngxStopCmdlistSave()` function to stop saving a command list, and get the command cache information.

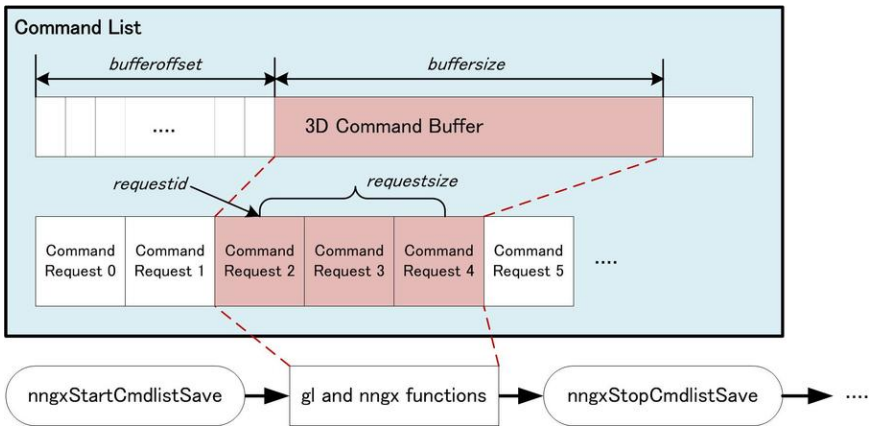
**Code 8-2. Stop Saving a Command List**

```
void nngxStopCmdlistSave(GLuint* bufferoffset, GLsizei* buffersize,
        GLuint* requestid, GLsizei* requestsize);
```

The *bufferoffset* parameter returns the offset at which saving of the 3D command buffer started, *buffersize* returns the save size (in bytes) of the 3D command buffer, *requestid* returns the ID at which saving of command requests started, and *requestsize* returns the number of saved command requests. The `GL_ERROR_8036_DMP` error is issued if the function is called before the command list starts saving.

The start and end of saving a command list may generate dummy commands as padding in the 3D Command Buffer of the command list.

**Figure 8-1. Saving Command Lists**




## 8.1.1. States

A state refers to all of the settings used by a particular 3D graphics feature. When you call a `gl()`

function or another graphics function, the state corresponding to that function is updated, and the 3D commands generated by the state update are accumulated in the 3D command buffer. Consequently, state updates have an effect on whether the 3D commands required for rendering are saved, and on whether command caches will function properly.

Each state has at least one `gl()` function or uniform setting. It is also possible that a single `gl()` function will update multiple states. If you specify states in the application (one reason to do this is to ensure that the required 3D commands are saved), you must specify states as a bitwise OR of the state flags.

**Table 8-1. State Flags**

| State Flag | Description |
|---|---|
| `NN_GX_STATE_SHADERBINARY` | Shader binary state.<br>Updated when you use the `glUseProgram()` function to switch to a different program object, and the shader objects linked to the new program object were not loaded by the same call to `glShaderBinary` that loaded the shader objects for the previous program object.<br>Updating this state generates 3D commands to load shader assembly code. |
| `NN_GX_STATE_SHADERPROGRAM` | Shader program state.<br>Updated when you use the `glUseProgram()` function to switch to a different program object.<br>Updating this state generates various 3D commands, including commands that set the configuration of vertex attributes. 3D commands are only generated for registers of settings that are changed since the previous validation. |
| `NN_GX_STATE_SHADERMODE` | Shader mode state.<br>Updated when you use the `glUseProgram()` function to enable or disable geometry shaders.<br>Updating this state generates 3D commands to enable or disable the use of the geometry shader. |
| `NN_GX_STATE_SHADERFLOAT` | Shader floating-point definition state.<br>Updated when you use the `glUseProgram()` function to switch to a program object with different shader objects attached.<br>Updating this state generates 3D commands to set floating-point registers with the values defined by the shader assembly `def` instructions. |
| `NN_GX_STATE_VSUNIFORM` | Vertex shader uniform state.<br>Updated when you use the `glUseProgram()` function to switch to a different program object, and also when a `glUniform*()` function sets the value of a vertex shader uniform. When integer uniforms are set, this state is updated only when their settings change. When floating-point uniforms are set, this state is updated whether or not the new settings are different from the old settings.<br>Updating this state generates 3D commands to set values in the floating-point registers, Boolean registers, and integer registers that are defined as uniforms in the shader assembly code. |
| `NN_GX_STATE_FSUNIFORM` | Reserved fragment shader uniform state.<br>Updated when you use the `glUseProgram()` function to switch to a different program object, and also when you use a `glUniform*()` function to set the value of a reserved fragment shader uniform.<br>Updating this state generates 3D commands to set values in the reserved fragment shader registers. |
| `NN_GX_STATE_LUT` | Lookup table state.<br>Updated when you use the `glBindTexture`, `glTexImage1D`, or `glTexSubImage1D()` functions to change the content of a lookup table, when you use the `glDeleteTextures()` function to delete a bound lookup table, and when you use the `glUseProgram` or |

| | |
|---|---|
| | glUniform∗() function to change the values of any uniforms associated with lookup tables.<br>Updating this state generates 3D commands to set the lookup tables. |
| NN_GX_STATE_TEXTURE | Texture state (excluding procedural textures).<br>Updated when any of the glBindTexture, glTexImage2D, glCopyTexImage2D, glCopyTexSubImage2D, glCompressedTexImage2D, or glTexParameter∗() functions are called, when the glDeleteTextures() function is used to delete a bound texture, and when the glUseProgram or glUniform∗() function is used to change the value of any of the dmp_Texture[*i*].samplerType reserved fragment shader uniforms.<br>Updating this state generates 3D commands involved with texture units (except for procedural textures). |
| NN_GX_STATE_FRAMEBUFFER | Framebuffer information state.<br>Updated when any of the glBindFramebuffer, glBindFramebufferRenderbuffer, glFramebufferTexture2D, glDeleteFramebuffers, glBindRenderbuffer, glRenderbufferStorage, or glDeleteRenderbuffers() functions are called.<br>Updating this state generates 3D commands involved with the framebuffer format and buffer address. |
| NN_GX_STATE_VERTEX | Vertex attribute data state.<br>Updated when any of the glBindBuffer, glBufferData, glBufferSubData, glEnableVertexAttribArray, glDisableVertexAttribArray, glVertexAttrib, glVertexAttribPointer, or glUseProgram() functions are called, and when the glDeleteBuffers() function is used to delete a bound vertex buffer.<br>Updating this state generates 3D commands involved with vertex attribute data. |
| NN_GX_STATE_TRIOFFSET | Polygon offset state.<br>Updated when you use the glEnable or glDisable() function to change the GL_POLYGON_OFFSET_FILL setting, when you use the glDepthRangef or glPolygonOffset() function to change settings, and when the glUseProgram() function is called.<br>Updating this state generates 3D commands involved with polygon offsets. |
| NN_GX_STATE_FBACCESS | Framebuffer access method state.<br>Updated when you use the glEnable or glDisable() function to change the GL_COLOR_LOGIC_OP, GL_BLEND, GL_DEPTH_TEST, GL_STENCIL_TEST, or GL_EARLY_DEPTH_TEST_DMP settings, when you use the glDepthFunc, glEarlyDepthFuncDMP, glColorMask, glDepthMask, or glStencilMask() function to change settings, and when you use a glUniform∗() function to set the dmp_FragOperation.mode reserved fragment shader uniform.<br>Updating this state generates 3D commands involved with framebuffer access methods (R/W, and so forth). |
| NN_GX_STATE_SCISSOR | Scissor test state.<br>Updated when you use the glEnable or glDisable() function to change the GL_SCISSOR_TEST setting, when you use the glScissor or glViewport() function to change settings, and when the framebuffer size is changed, while the scissor test is enabled.<br>Updating this state generates 3D commands involved with the scissor test. |
| NN_GX_STATE_OTHERS | State of all functions that generate 3D commands besides glDraw∗() functions.<br>Updated when functions other than glDrawElements or glDrawArrays are called that generate 3D commands. |

| | |
|---|---|
| | For more information about the 3D commands generated when this state is updated, see Table 8-3. |
| NN_GX_STATE_ALL | This flag specifies all of the above states. |

## 8.1.2. Generating Commands

Most of the 3D commands accumulated in 3D Command Buffers are generated by the glDrawElements() and glDrawArrays() functions. When these functions run, they check the state and generate the 3D commands involved with any state updates. This is known as validating.

States can be validated at any time by calling the nngxValidateState() function. Alternatively, you can specify state flags to the nngxUpdateState() function to mark those states as updated. By doing so, you can generate all 3D commands for those specified states, at a time of your choosing.

**Code 8-3. Validating State and Specifying State Flags**

```
void nngxValidateState(GLbitfield statemask, GLboolean drawelements);
void nngxUpdateState(GLbitfield statemask);
```

Out of all the updated states, calling the nngxValidateState() function only validates the states specified by *statemask*, and then removes these states from updated status. This function does not cause any rendering to take place, so specify in the *drawelements* parameter whether you will use the glDrawElements() or the glDrawArrays() function when actually rendering. Pass GL_TRUE to generate 3D commands corresponding to the glDrawElements() function, and pass GL_FALSE to generate 3D commands for the glDrawArrays() function. In conclusion, you can use a combination of the nngxUpdateState() and nngxValidateState() functions to generate all the 3D commands for any state.

In contrast, glDrawElements() and the other rendering functions cause all updated states to be validated and then removed from "updated" status.

Unlike other functions, the nngxValidateState() function can generate 3D commands at any time. Note, however, that there is a prescribed order in which the various states must be updated. The 3D commands for some states must be set before the 3D commands for other states. This prescribed order is described below. Operation may become unstable if 3D commands are generated and run in any order other than as described by the following conditions.

- NN_GX_STATE_FBACCESS and NN_GX_STATE_TRIOFFSET must be specified before, or at the same time as, NN_GX_STATE_FSUNIFORM.
- NN_GX_STATE_SHADERMODE must be specified before, or at the same time as, NN_GX_STATE_SHADERBINARY, NN_GX_STATE_SHADERPROGRAM, NN_GX_STATE_SHADERFLOAT, and NN_GX_STATE_VSUNIFORM.
- NN_GX_STATE_FRAMEBUFFER and NN_GX_STATE_OTHERS must be specified before, or at the same time as, NN_GX_STATE_FBACCESS.
- NN_GX_STATE_FRAMEBUFFER must be specified before, or at the same time as, NN_GX_STATE_SCISSOR.

**Table 8-2. Common Errors When Using the nngxValidateState Function**

| Error | Cause |
|---|---|
| | |

| | |
|---|---|
| GL_ERROR_8066_DMP | Calling the nngxValidateState() function resulted in a 3D Command Buffer overflow. |
| GL_ERROR_806C_DMP | An error occurred during validation. |
| GL_ERROR_80B2_DMP | The 3D Command Buffer is not set. |
| GL_ERROR_80B3_DMP | Without setting a valid program object (glUseProgram has never been called, or 0 was specified for program when glUseProgram was called), you attempted to validate states related to the *program* object as in the following list.<br>• NN_GX_STATE_SHADERBINARY<br>• NN_GX_STATE_SHADERPROGRAM<br>• NN_GX_STATE_SHADERMODE<br>• NN_GX_STATE_SHADERFLOAT<br>• NN_GX_STATE_VSUNIFORM<br>• NN_GX_STATE_FSUNIFORM<br>• NN_GX_STATE_LUT<br>• NN_GX_STATE_VERTEX<br>• NN_GX_STATE_TRIOFFSET<br>• NN_GX_STATE_FBACCESS |

The following are causes of validation errors.

- No texture memory allocated for the enabled textures.
  To resolve this error, you must call the glTexImage2D, glCompressedTexImage2D, or glCopyTexImage2D() function and allocate texture memory. For cube map textures, memory must be allocated for all six faces.
- A bound texture is in an invalid format.
  This could be caused, for example, when a texture of format GL_SHADOW_DMP is bound to texture unit 1 or 2, or when a texture of format GL_GAS_DMP is bound to a cube map texture.
- The settings for the faces of a cube map texture differ.
  The six faces of a cube map texture must all have the same width, height, format, and mipmap level.
- The most-significant seven bits of the addresses for all six faces of a cube map texture do not match.
  The most-significant seven bits of the addresses for all six faces of a cube map texture must be the same.
- A lookup table object is not properly bound, or a lookup table number is not properly specified.
  When lookup tables are configured for use in fragment lighting, procedural textures, fog, or gas, valid lookup table objects must be bound to the relevant lookup table numbers. The uniforms that specify the lookup table numbers must also be properly configured.
- Failed to allocate the memory required to store the lookup tables' internal format values.

If an error occurred during validation, each state is considered to be validated. To generate the correct command after an error, call the nngxUpdateState() function and update the state again.

Generally, 3D commands are generated by the glDraw() functions, but 3D commands for NN_GX_STATE_FRAMEBUFFER state updates can also be generated by the glReadPixels() and glClear() functions. In addition to those, calling the following functions will also generate 3D commands.

**Table 8-3. Functions That Generate 3D Commands**

| | |
|---|---|

| Function | When Generated |
|---|---|
| glBlendColor | When a setting value has been changed. |
| glBlendEquation | When a setting value has been changed. |
| glBlendEquationSeparate | When a setting value has been changed. |
| glBlendFunc | When a setting value has been changed. |
| glBlendFuncSeparate | When a setting value has been changed. |
| glClearEarlyDepthDMP | When a setting value has been changed. |
| glColorMask | When a setting value has been changed. |
| glCullFace | When a setting value has been changed. |
| glDepthFunc | When a setting value has been changed. |
| glDepthMask | When a setting value has been changed. |
| glDisable | When the GL_COLOR_LOGIC_OP, GL_BLEND, GL_DEPTH_TEST, GL_EARLY_DEPTH_TEST_DMP, GL_STENCIL_TEST, or GL_CULL_FACE setting has been changed. No commands generated if other setting values have changed. |
| glEarlyDepthFuncDMP | When a setting value has been changed. |
| glEnable | When the GL_COLOR_LOGIC_OP, GL_BLEND, GL_DEPTH_TEST, GL_EARLY_DEPTH_TEST_DMP, GL_STENCIL_TEST, or GL_CULL_FACE setting has been changed. No commands generated if other setting values have changed. |
| glFrontFace | When a setting value has been changed. |
| glLogicOp | When a setting value has been changed. |
| glRenderBlockModeDMP | When a setting value has been changed. |
| glStencilFunc | When a setting value has been changed. |
| glStencilMask | When a setting value has been changed. |
| glStencilOp | When a setting value has been changed. |
| glViewport | Always generated. |

Of the functions in this table, the glEnable, glDisable, glDepthFunc, glEarlyDepthFuncDMP, glColorMask, glDepthMask, and glStencilMask() functions all update the NN_GX_STATE_FBACCESS state when they are called for certain features (GL_COLOR_LOGIC_OP, GL_BLEND, GL_DEPTH_TEST, GL_STENCIL_TEST, and GL_EARLY_DEPTH_TEST_DMP), so you must validate the NN_GX_STATE_FBACCESS state when you save a command list.

If you specified NN_GX_STATE_OTHERS to the nngxUpdateState() function, 3D commands for all the functions in the table above are generated at state validation time. The same function call also marks the NN_GX_STATE_FBACCESS state as updated.

## 8.1.3. Commands for Partial State Updates and Full State Updates

The gl() functions may update multiple states. Normally, a called function only generates 3D commands for updated states. This is known as a *partial state update*, and these commands are *delta commands*. Meanwhile, a full state update is when the called function generates 3D commands for all associated states, regardless of whether they have been updated. Full state updates may lead to redundant 3D command generation because they cause all 3D commands for a state to be generated separately again for each state.

Use the nngxSetCommandGenerationMode() function to configure several states to always trigger full state updates. You can also use the nngxUpdateState() function to specify a full state update of the specified states.

**Code 8-4. Setting the Command Generation Mode**

```
void nngxSetCommandGenerationMode(GLenum mode);
```

Specify NN_GX_CMDGEN_MODE_CONDITIONAL in the mode parameter to generate delta commands. This is the default mode. Specify NN_GX_CMDGEN_MODE_UNCONDITIONAL to generate all commands for a full state update. GL_ERROR_804D_DMPAn invalid value was specified in the *mode* parameter. Call the nngxGetCommandGenerationMode() function to get the current mode.

**Code 8-5. Getting the Command Generation Mode**

```
void nngxGetCommandGenerationMode(GLenum* mode);
```

The current mode is returned in the *mode* parameter.

The following states are affected by unconditional command generation mode (which specifies full state updates).

- States involved with setting reserved fragment shader uniforms.
- States involved with setting vertex shader integer uniforms.
- States involved with setting lookup table data.
- States involved with the functions listed in Table 8-3.

In addition to the 3D commands for updated states, the states in this list have all commands generated for a full state update, regardless of whether they have been updated. However, even if all states involved with lookup tables have been updated, only those 3D commands for the enabled lookup tables are generated. The following lists the conditions under which a lookup table is enabled.

**Table 8-4. Conditions for a Lookup Table to Be Enabled**

| Lookup Table | Conditions |
|---|---|
| Red component of reflection (RR) in fragment lighting | dmp_FragmentLighting.enabled is set to GL_TRUE, dmp_LightEnv.config is set to use RR, and dmp_LightEnv.lutEnabledRefl is set to GL_TRUE. |
| Green component of reflection (RG) in fragment lighting | dmp_FragmentLighting.enabled is set to GL_TRUE, dmp_LightEnv.config is set to use RG, and dmp_LightEnv.lutEnabledRefl is set to GL_TRUE. |
| Blue component of reflection (RB) in fragment lighting | dmp_FragmentLighting.enabled is set to GL_TRUE, dmp_LightEnv.config is set to use RB, and dmp_LightEnv.lutEnabledRefl is set to GL_TRUE. |
| Distribution factor 0 (D0) in fragment lighting | dmp_FragmentLighting.enabled is set to GL_TRUE, dmp_LightEnv.config is set to use D0, and dmp_LightEnv.lutEnabledD0 is set to GL_TRUE. |
| Distribution factor 1 (D1) in fragment lighting | dmp_FragmentLighting.enabled is set to GL_TRUE, dmp_LightEnv.config is set to use D1, and dmp_LightEnv.lutEnabledD1 is set to GL_TRUE. |

| | |
|---|---|
| Fresnel factor (FR) in fragment lighting | `dmp_FragmentLighting.enabled` is set to `GL_TRUE`, `dmp_LightEnv.config` is set to use FR, and `dmp_LightEnv.fresnelSelector` is set to a value other than `GL_LIGHT_ENV_NO_FRESNEL_DMP`. |
| Spotlight attenuation (SP) in fragment lighting | `dmp_FragmentLighting.enabled` is set to `GL_TRUE`, `dmp_LightEnv.config` is set to use SP, `dmp_FragmentLightSource[`*i*`].enabled` is set to `GL_TRUE`, and `dmp_FragmentLightSource[`*i*`].spotEnabled` is set to `GL_TRUE`. |
| Distance attenuation in fragment lighting | `dmp_FragmentLighting.enabled` is set to `GL_TRUE`, `dmp_FragmentLightSource[`*i*`].enabled` is set to `GL_TRUE`, and `dmp_FragmentLightSource[`*i*`].distanceAttenuationEnabled` is set to `GL_TRUE`. |
| RGB-mapping F function in procedural textures | `dmp_Texture[3].samplerType` is set to `GL_TEXTURE_PROCEDURAL_DMP`. |
| Alpha-mapping F function in procedural textures | `dmp_Texture[3].samplerType` is set to `GL_TEXTURE_PROCEDURAL_DMP` and `dmp_Texture[3].ptAlphaSeparate` is set to `GL_TRUE`. |
| Noise modulation function in procedural textures | `dmp_Texture[3].samplerType` is set to `GL_TEXTURE_PROCEDURAL_DMP`, and `dmp_Texture[3].ptNoiseEnable` is set to `GL_TRUE`. |
| Color lookup table in procedural textures | `dmp_Texture[3].samplerType` is set to `GL_TEXTURE_PROCEDURAL_DMP`. |
| Fog coefficient | `dmp_Fog.mode` is set to a value other than `GL_FALSE`. |
| Shading lookup table for gas | `dmp_Fog.mode` is set to `GL_GAS_DMP`. |

## 8.1.4. Command Cache Restrictions and Notes

Command cache use has the following restrictions and notes.

- When fragment lighting is enabled (reserved uniform `dmp_FragmentLighting.enabled` is set to `GL_TRUE`) and all light sources are disabled (reserved uniforms `dmp_FragmentLightSource[`*i*`].enabled` are all set to `GL_FALSE`), 3D commands for lighting are generated again when a rendering function is called, even after you have specified the state flag for reserved fragment shader uniforms (`NN_GX_STATE_FSUNIFORM`) to the `nngxValidateState()` function and validated the state.
- 3D commands for the `dmp_Gas.accMax` reserved uniform are generated again when a rendering function is called, even after you have specified the state flag for reserved fragment shader uniforms (`NN_GX_STATE_FSUNIFORM`) to the `nngxValidateState()` function and validated the state.
- When the reserved fragment uniform `dmp_Gas.autoAcc` is set to `GL_TRUE`, either starting saving or stopping saving of a command list, and when the value of the reserved uniform `dmp_FragOperation.mode` changes to `GL_FRAGOP_MODE_GAS_ACC_DMP`, or when it changes from `GL_FRAGOP_MODE_GAS_ACC_DMP` to another value, this sometimes results in the wrong commands for `dmp_Gas.autoAcc` in the saved command list.
- To run a 3D command buffer, its size must be a multiple of 16 bytes. Adjust the size by calling the `nngxAdd3DCommand()` function and appending the filler value `0x00000000_00000000`.
- If you call the `glUseProgram()` function and pass `0` as an argument, no commands are

generated during validation of states involved with programs or shaders.

### 8.1.5. Getting Updated States

Call the `nngxGetUpdateState()` function to get the state flags of states that have been updated.

**Code 8-6. Getting Updated States**

```
void nngxGetUpdatedState (GLbitfield* statemask);
```

### 8.1.6. Disabling State Updates

Call the `nngxInvalidateState()` function to disable state updates.

**Code 8-7. Disabling State Updates**

```
void nngxInvalidateState (GLbitfield statemask);
```

For the *`statemask`* parameter, specify the bitwise OR of the state flags of states for which to disable updates. Thereafter, no commands are generated for the states whose state flags are specified in *statemask*, even if these states are updated.

## 8.2. Reusing Command Lists

You can reuse 3D commands and Command Requests by taking the command cache information obtained from saving a command list and passing it to the `nngxUseSavedCmdlist` or `nngxUseSavedCmdlistNoCacheFlush()` functions. The former flushes the cache for the added 3D commands, but the latter does not.

**Code 8-8. Reusing a Command List**

```
void nngxUseSavedCmdlist(GLuint cmdlist, GLuint bufferoffset,
        GLsizei buffersize, GLuint requestid, GLsizei requestsize,
        GLbitfield statemask, GLboolean copycmd);
void nngxUseSavedCmdlistNoCacheFlush(GLuint cmdlist,
        GLuint bufferoffset, GLsizei buffersize,
        GLuint requestid, GLsizei requestsize, GLbitfield statemask);
```

The *cmdlist* parameter specifies the command list that was saved and in which 3D commands are accumulated.

The *bufferoffset* parameter specifies the offset at which saving of the 3D command buffer started, the *buffersize* parameter specifies the 3D command buffer's save size in bytes, the *requestid* parameter

specifies the ID at which saving of command requests started, and the *requestsize* parameter specifies the number of saved command requests. These must all be from the same command cache or from command caches that were saved at the same time. The function does not check to confirm that this command cache information is actually from the specified command list, or that all this information is from the same command cache. If you specify incorrect command cache information, resulting operations are undefined.

The *statemask* parameter specifies a bitwise OR of the state flags of states for which to do full state updates. When you call this function, an inconsistency arises between state settings before and after the function call. To resolve this inconsistency it is sometimes necessary to generate commands for full state updates of all states. But because generating all commands for all states can be excessive, full state updates are performed only for those states specified in *statemask*. If NN_GX_STATE_OTHERS is specified in *statemask*, commands for full state updates are generated for all states involved with the functions listed in Table 8-5.

The *copycmd* parameter specifies whether the command cache feature first copies the 3D commands from the saved command list before running them, or just runs them directly from the saved command list. If GL_TRUE is specified, the 3D commands are copied and appended to the currently bound command list. Copying the 3D commands entails a high CPU load, so this method is most appropriate when reusing small 3D command buffers. If GL_FALSE is specified, the commands are not copied. Not copying the 3D commands reduces the CPU load, so this method is most appropriate when reusing large 3D command buffers. This parameter only controls whether 3D commands are copied and added to the 3D command buffer. Command requests are always added.

If you do not copy the 3D commands from the saved command list and the 3D commands are not already split, this function adds a "split" command immediately prior to the reused 3D commands and switches to a command list that has saved the 3D command execution address. Consequently, if there is no split command in the saved command list, it is not possible to return to the original command list. To reuse commands from a command list saved this way, you must always add a split command using the nngxSplitDrawCmdlist() function before you stop saving the command list.

If you copy the 3D commands from the saved command list under the following conditions, this function adds a split command immediately prior to the reused 3D commands.

The operation of the nngxUseSavedCmdlistNoCacheFlush() function is the same as when designating the *copycmd* of nngxUseSavedCmdlist() as GL_FALSE, but it does not perform a cache flush of the 3D command buffer of the command list specified with *cmdlist*. The application must ensure that the 3D command buffer region has been flushed.

**Figure 8-2. Difference in Operation of nngxUseSavedCmdlist() due to copycmd**

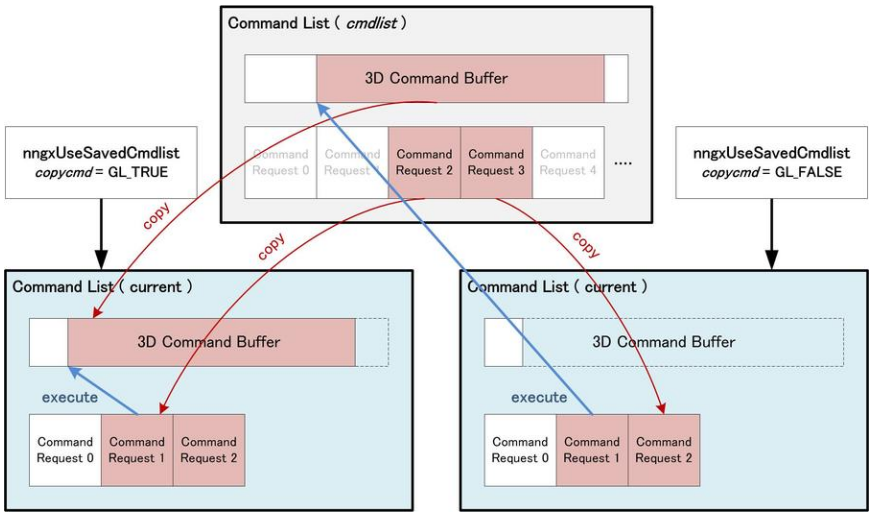


**Table 8-5. Common Errors When Using the nngxUseSavedCmdlist() and the nngxUseSavedCmdlistNoCacheFlush() Functions**

| Error | Cause |
|---|---|
| GL_ERROR_8037_DMP<br>GL_ERROR_8092_DMP | Called when the bound command list's object name is `0`. |
| GL_ERROR_8038_DMP<br>GL_ERROR_8093_DMP | A nonexistent command list was passed in *cmdlist*. |
| GL_ERROR_803A_DMP<br>GL_ERROR_8095_DMP | Running the function causes the currently bound command list's 3D command buffer to overflow or its stored command requests to exceed the maximum number. |

## 8.2.1. Command Request Information That Is Copied

When the `nngxUseSavedCmdlist()` function runs, Command Requests are always copied and appended to the currently bound command list, regardless of the value passed in *copycmd*. Command Requests contain information that varies depending on the type of Command Request, and this information is copied unchanged, even if states have been updated since the command list was saved. However, this does not apply to the first render command request that is copied. It is possible for this command's information to change.

#### DMA Transfer Command Requests

These requests contain the DMA transfer source address, destination address, and transfer size.

#### Render Command Requests

These requests contain the 3D command buffer's execution start address and execution size. When the address of the 3D command buffer for which saving has begun does not match the execution start address, the execution start address is replaced with the address at which saving started when the command requests were copied, and the execution size is changed to match.

#### Memory-Fill Command Requests

These requests contain the starting address, size, and clear color of the color buffer to fill, in addition to the starting address, size, clear depth value, and clear stencil value of the depth stencil buffer.

#### Post-Transfer Command Requests

These contain the address, resolution, and format of the color buffer that is the transfer source, along with the address, resolution, and format of the destination display buffer.

#### Copy Texture Command Requests

These contain the address and resolution of the color buffer that is the transfer source, along with the address and resolution of the destination texture.

# 8.3. Copying Command Lists

Call the `nngxCopyCmdlist()` function to copy the contents of one command list to a different command list. Note that this function copies all of the command list information and overwrites any accumulated 3D commands and command requests in the destination command list.

**Code 8-9. Copy-Appending a Command List**

```
void nngxCopyCmdlist(GLuint scmdlist, GLuint dcmdlist);
```

The *`scmdlist`* parameter specifies the source command list to copy from, and the *`dcmdlist`* parameter specifies the destination where the command list is to be copied.

The operations of this function do not directly relate to command caches, but if that command cache information is created based on an offset, you can use command caches to reuse a copy created right after saving a command list. You can also clear the copy source command list after copying.

**Table 8-6. Common Errors When Using the nngxValidateState Function**

| Error | Cause |
|---|---|
| `GL_ERROR_8047_DMP` | The currently bound command list was specified for *dcmdlist*. |
| `GL_ERROR_8048_DMP` | A nonexistent command list was passed in *cmdlist*. |
| `GL_ERROR_8049_DMP` | A nonexistent command list was passed in *cmdlist*. |
| `GL_ERROR_804A_DMP` | The same command list was specified for both *scmdlist* and *dcmdlist*. |
| `GL_ERROR_804B_DMP` | A running command list was specified for *dcmdlist*. |
| `GL_ERROR_804C_DMP` | A command list was specified for *scmdlist* that has accumulated more 3D commands or Command Requests than can fit in the command list specified for *dcmdlist*. |

## 8.3.1. Copying Command Lists

The `nngxCopyCmdlist()` function only supports copying a command list and using it to overwrite the destination list, but the `nngxAddCmdList()` function allows you to copy all the information in a command list and append it to the currently bound command list.

**Code 8-10. Copy-Appending a Command List**

```
void nngxAddCmdlist(GLuint cmdlist);
```

The *`cmdlist`* parameter specifies the source command list to copy.

All of the commands accumulated in the source command list are appended to the currently bound command list. The copied commands are added to the end of the currently bound command list, after any commands it has already accumulated.

If the currently bound command list's 3D command buffer has not just been split, and if the first command in the command requests that are being appended is not a render command request, the library first calls the `nngxSplitDrawCmdlist()` function to split the 3D Command Buffer, and then appends the copied commands.

If the currently bound command list's 3D command buffer has not just been split, and if the first command in the appended command requests is a render command request, dummy commands are added to the destination 3D command buffer as needed to align it before the copied commands are appended.

If the library must call `nngxSplitDrawCmdlist()` or append additional dummy commands, the commands added by that processing are included in this maximum size check.

**Table 8-7. Common Errors When Using the nngxValidateState Function**

| Error | Cause |
|---|---|
| `GL_ERROR_8054_DMP` | An invalid value was specified for *cmdlist*. |
| `GL_ERROR_8055_DMP` | There is no currently bound command list. |
| `GL_ERROR_8056_DMP` | The currently bound command list was specified in *cmdlist*. |
| `GL_ERROR_8057_DMP` | The currently bound command list is running. |
| `GL_ERROR_8058_DMP` | A command list was specified that has a 3D Command Buffer or Command Requests that are too big to fit in the currently bound command list. |

## 8.4. Exporting Command Lists

Call the `nngxExportCmdlist()` function to store the contents of a command list, obtained via a command cache, in memory as binary data. This operation is equivalent to exporting the command list.

**Code 8-11. Exporting a Command List**

```
GLsizei nngxExportCmdlist(GLuint cmdlist, GLuint bufferoffset,
        GLsizei buffersize, GLuint requestid, GLsizei requestsize,
        GLsizei datasize, GLvoid* data);
```

The *cmdlist* parameter specifies the command list to export.

The *bufferoffset* and *buffersize* parameters specify the byte offset and byte size of the 3D Command Buffer memory region to export. The *requestid* and *requestsize* parameters specify the Command Request ID at which to start exporting (these IDs start from 0 in the order of accumulation), and the number of Command Requests to export.

The *data* and *datasize* parameters specify the starting address and size of the export destination memory region. This function returns the size, in bytes, of the exported data, but if the *data* parameter specifies `0` or `NULL`, no data is exported. Instead the function returns the memory size required for export. The export procedure is to first get the size of the memory needed, allocate a memory region, and then finally export the data.

This function's *bufferoffset*, *buffersize*, *requestid*, and *requestsize* parameters must specify values that are not mutually contradictory. To safely export data, we recommend using save information obtained from the `nngxStopCmdlistSave()` function, or values obtained by several carefully timed calls of the `nngxGetCmdlistParameteri()` function during 3D command accumulation. Pass `NN_GX_CMDLIST_USED_BUFSIZE` to the *pname* of `nngxGetCmdlistParameteri()` to get the accumulated 3D command buffer size, and pass `NN_GX_CMDLIST_USED_REQCOUNT` to get the number of accumulated command requests. These two values are obtained together at the start and end of accumulation of the command list to be exported. Specify the values obtained at the start to

*bufferoffset* and *requested*, and subtract the start values from the end values and specify the resulting values to *buffersize* and *requestsize*.

The 3D command buffer region to be exported must fulfill the following conditions.

- The `bufferoffset` value must point somewhere within the memory region executed by the first render command request, in the command requests to export.
- All of the split commands executed by the exported render command requests must also be exported.
- The region after the region where the last 3D execution command was executed can also be exported, but that region must not contain any split commands.
- If no command requests are exported, split commands must not be included in the exported commands.

**Figure 8-3. Exportable 3D Command Buffer Regions**

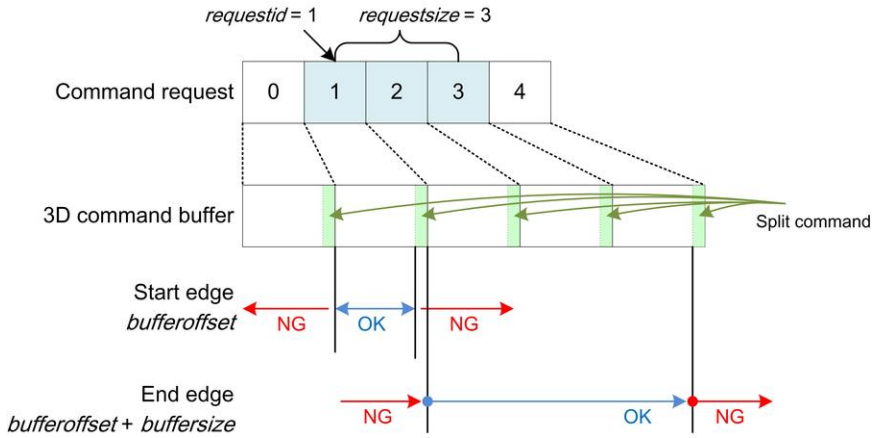



**Figure 8-4. Exportable 3D Command Buffer Regions (When No Command Requests Are Exported)**

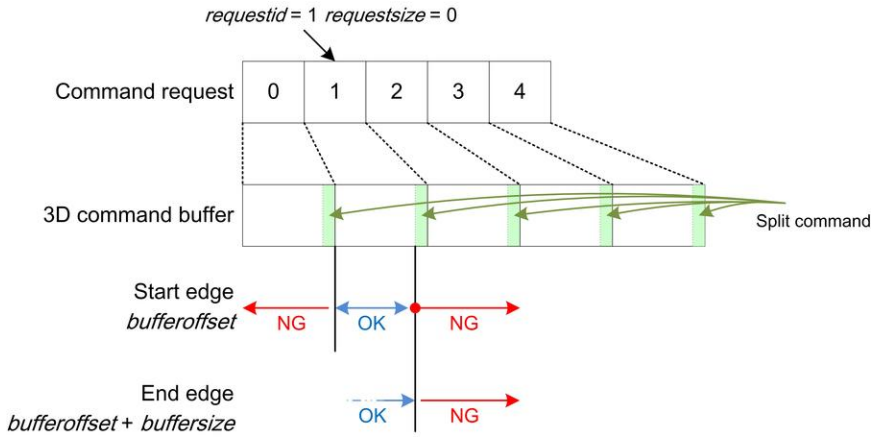



**Table 8-8. Common Errors When Using the nngxExportCmdlist() Function**

| Error | Cause |
|---|---|
| `GL_ERROR_803B_DMP` | An invalid value (`0` or a nonexistent command list) was specified. |
| `GL_ERROR_803C_DMP` | A value was specified for the *datasize* parameter that is smaller than the size of the data to export. |
| `GL_ERROR_803D_DMP` | The region specified for *requestid* and *requestsize* does not have any commands accumulated. |

| | |
|---|---|
| GL_ERROR_803E_DMP | The *bufferoffset* or *buffersize* values specified are not 8-byte aligned. |
| GL_ERROR_803F_DMP | A command list was specified that includes a render command request added in a method other than the nngxUseSavedCmdlist() function copying the command request. |
| GL_ERROR_8040_DMP | Incorrect 3D Command Buffer values were specified for *bufferoffset* and *buffersize* for the exported render command request that executes the 3D command. |

### 8.4.1. Getting the Export Information

Use the nngxGetExportedCmdlistInfo() function to get the command list information (export information) that is included in exported binary data.

**8-12. Getting the Export Information**

```
void nngxGetExportedCmdlistInfo(GLvoid* data, GLsizei* buffersize,
        GLsizei* requestsize, GLuint* bufferoffset);
```

The *data* parameter specifies the starting address of the exported binary data. Specifying invalid data causes a GL_ERROR_8046_DMP error. There are four parameters in the export information: *buffersize* stores the size, in bytes, of the 3D Command Buffer, *requestsize* stores the number of Command Requests, and *bufferoffset* stores the byte offset from the start of the exported data, specified in *data*, to the start of the 3D Command Buffer.

## 8.5. Importing Command Lists

Call the nngxImportCmdlist() function to copy-append 3D commands from exported binary data to a command list. This operation is equivalent to importing an exported list.

**Code 8-13. Importing a Command List**

```
void nngxImportCmdlist(GLuint cmdlist, GLvoid* data, GLsizei datasize);
```

The *cmdlist* parameter can specify either the currently bound command list or a command list that is not bound. If the specified command list has already accumulated 3D commands, the imported commands will be appended after the accumulated ones.

If the first command request in the data to import is not a render command request, you must bind the destination command list, and then add a split command to it using the nngxSplitDrawCmdlist() function.

The *data* and *datasize* parameters specify a pointer to the export data and the size, in bytes, of the export data.

Importing a command list may cause dummy commands to be generated as padding in the 3D Command Buffer of the destination command list.

**Table 8-9. Common Errors When Using the nngxExportCmdlist Function**

| Error | Cause |
|---|---|
| GL_ERROR_8041_DMP | An invalid value (0 or a nonexistent command list) was specified. |
| GL_ERROR_8042_DMP | An invalid pointer was specified for the *data* parameter. |
| GL_ERROR_8043_DMP | A size that is different from the export data size was specified for the *datasize* parameter. |
| GL_ERROR_8044_DMP | Importing more 3D commands or Command Requests than can fit within the size of the destination command list. |
| GL_ERROR_8045_DMP | Data that does not have a render command request as its first Command Request is being imported into an unsplit command list. |

## 8.6. Adding 3D Commands

Use the nngxAdd3DCommand or nngxAdd3DCommandNoCacheFlush() function either to specify a region and add the data in that specified region to the 3D Command Buffer of the currently bound command list, or to add render command requests that run the specified region. The former flushes the cache for the added 3D commands, but the latter does not.

**8-14. Adding 3D Commands**

```
void nngxAdd3DCommand(const GLvoid* bufferaddr, GLsizei buffersize,
        GLboolean copycmd);
void nngxAdd3DCommandNoCacheFlush(const GLvoid* bufferaddr,
        GLsizei buffersize);
```

This function works differently depending on the value of the *copycmd* parameter.

When *copycmd* is set to GL_TRUE, this function copy-appends the 3D commands stored in the region whose starting address and size, in bytes, are specified by *bufferaddr* and *buffersize* to the 3D Command Buffer of the currently bound command list. Operation is not guaranteed if the specified region contains split commands. The *buffersize* value must be a positive multiple of 4.

When *copycmd* is set to GL_FALSE, this function adds a render command request that runs the 3D commands stored in the region whose starting address and size, in bytes, are specified by *bufferaddr* and *buffersize* to the Command Requests. If the 3D Command Buffer of the currently bound command list is unsplit, this function adds a split command, and then adds the Command Request. Operation is not guaranteed if the last 3D command in the specified region is not a split command. The *bufferaddr* value must be a multiple of 16 and the *buffersize* value must be a positive multiple of 16.

The operation of the nngxAdd3DCommandNoCacheFlush() function is the same as when designating the *copycmd* of nngxAdd3DCommand() as GL_FALSE.

**Table 8-10. Common Errors When Using the nngxAdd3DCommand() and the nngxAdd3DCommandNoCacheFlush() Functions**

| Error | Cause |
|---|---|
| GL_ERROR_804E_DMP<br>GL_ERROR_808C_DMP | Called without a bound command list. |
| GL_ERROR_804F_DMP<br>GL_ERROR_808D_DMP | An invalid value was specified for *buffersize*. |
| | The specified 3D Command Buffer size is insufficient for the currently bound |

| | |
|---|---|
| GL_ERROR_8050_DMP | command list, when *copycmd* is GL_TRUE. |
| GL_ERROR_8051_DMP | The specified 3D Command Buffer size is insufficient for the currently bound command list, when *copycmd* is GL_FALSE. |
| GL_ERROR_8052_DMP | A value that is not a multiple of 16 was specified for *bufferaddr*, when *copycmd* is GL_FALSE. |
| GL_ERROR_808E_DMP | A value that is not a multiple of 16 was specified for *bufferaddr*, when *copycmd* is GL_FALSE. |
| GL_ERROR_808F_DMP | The specified Command Request is insufficient for the currently bound command list. |

## 8.6.1. Directly Generating 3D Commands

You can use the nngxAdd3DCommand() function to run 3D commands that have been directly generated by the application. You can run these 3D commands without calling any gl() functions. Note, however, that when you run a mix of directly generated 3D commands and 3D commands generated by calling the gl() functions (regularly generated commands), the directly generated commands do not update any of the library states.

If you change GPU settings with directly generated commands, there is a possibility that this will not be recognized as a state update. (Comparing the current states to the states after calling gl() functions that change the same settings might not show that any update occurred.) If your changes are not recognized, 3D commands that are normally generated are not generated, leading to unintended rendering results. Likewise, if you run directly generated commands while some states are still marked as updated and then perform state validation, unintended 3D commands may be generated and prevent your directly generated commands from being applied.

### 8.6.1.1. Transitioning From Regularly Generated Commands to Directly Generated Commands

The key to safely transitioning from running regularly generated commands to running directly generated commands is to ensure that when you do so, no states are currently marked as "updated."

When a state has been marked as updated, that updated status is removed only after a glDraw() function or the nngxValidateState() function is called and the state is validated (see 8.1.2. Generating Commands). If you run directly generated commands on the assumption that all settings previously configured by gl() functions are already applied to the GPU, we strongly recommend validating states first. This makes your assumption true and also ensures that your directly generated commands are run in the way you expect. It is easy to validate all states by passing NN_GX_STATE_ALL in a call to the nngxValidateState() function. But to prevent the generation of excess 3D commands, you can instead call the nngxGetUpdatedState() function to get the state flags of the states marked as updated, and validate just those states.

### 8.6.1.2. Transitioning From Directly Generated Commands to Regularly Generated Commands

The key to safely transitioning from running directly generated commands to running regularly generated commands is to mark as updated all states that would have been updated by directly generated commands. Doing so ensures that the GPU settings and the library states match.

The most certain method is to mark all states as updated by passing `NN_GX_STATE_ALL` in a call to the `nngxValidateState()` function, and then validate all states. However, this method causes the generation of unneeded 3D commands.

If you thoroughly understand what states are updated by your directly generated commands and what states are dependent on those states, you can mark only those states as "updated," validate them, and keep the 3D commands generated to the minimum required.

If you pass `NN_GX_CMDGEN_MODE_UNCONDITIONAL` in a call to the `nngxSetCommandGenerationMode()` function and do full state updates of only some states, you can avoid excessive command generation. However, when you do so, the only states that are okay to not mark as "updated" with the `nngxUpdateState()` function are the specific states described in 8.1.3. Commands for Partial State Updates and Full State Updates.

# 8.7. Editing 3D Commands

Command lists can be reused by means of command caches. However, simply reusing a saved command list as is does not take into account any intervening changes in the scene, such as updates to the camera position. This section describes the 3D Command Buffer specifications and the information written to GPU registers. It also introduces how to handle changes in the scene by editing 3D commands, such as the 3D commands involved in vertex shader and reserved fragment shader settings.

## 8.7.1. 3D Command Buffer Access

Part of the information saved in a command cache is the offset at which saving of the 3D Command Buffer started. Consequently, to access a 3D Command Buffer to edit commands, you must first get the starting address of the 3D Command Buffer by calling the `nngxGetCmdlistParameteri()` function and passing `NN_GX_CMDLIST_TOP_BUFADDR` in the *pname* parameter.

3D commands and the values written to registers are little-endian, so you must pay careful attention to the correspondence between the byte layouts and numerical notations shown below and the byte order of this data in memory.

## 8.7.2. 3D Command Buffer Specifications

The 3D Command Buffer is a collection of 3D commands (PICA register write commands) for writing to the GPU registers. 3D commands are collected into continuous 64-bit segments, with 32 bits of header and 32 bits of data. The number of data items varies depending on the header content, but 3D commands are always 64-bit aligned. The upper 32 bits of the final 64-bit segment may be ignored, depending on the number of data items.

Data is stored in the individual 3D command bits as follows.

**Figure 8-5. 3D Command Bit Layout**

**Table 8-11. 3D Command Bit Descriptions**

| Bits | Name | Description |
|---|---|---|
| [31:0] | DATA | 32 bits of data to write to registers. |
| [47:32] | ADDR | PICA register address to which to write the data. |
| [51:48] | BE | Byte enabled. The 32-bit data segment is broken into 4 single-byte data items. A byte is written if its corresponding BE bit is 1. |
| [59:52] | SIZE | Data item count. The value stored is 1 less than the actual number of data items. Single access is used if this value is 0. Burst access is used if this value is 1 or greater. |
| [63:63] | SEQ | Access mode during burst access. If 0, all write operations are to a single register. If 1, multiple registers are written sequentially. |

If BE is 0, data will not be written to the register, but the 3D commands are written to the GPU, so it can be used for alignment or timing adjustment as a dummy command. However, note that there is a range of registers that can be specified in ADDR as dummy commands.

There are two access methods (single access and burst access), depending on the number of data items specified in SIZE. There are also two possible access modes during burst access (write single registers or write sequential sets of registers), depending on the value specified in SEQ.

## 8.7.3. Single Access

When SIZE is 0, there is only one data item and the data is written to a register using single access. With single access, one data item is written to one register only once.

The contents of DATA are written to the register specified in ADDR. Only those bytes for which the K bit value is 1 are written. Data is not written to the register if the corresponding BE value is 0. The SEQ value is ignored.

Example:

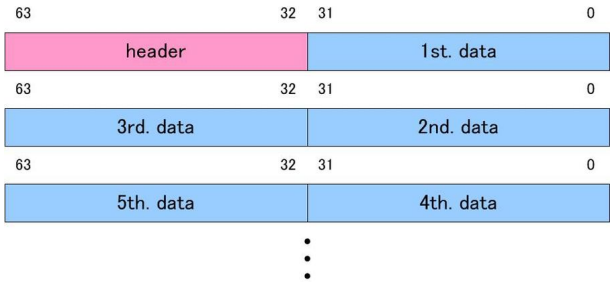
> A 3D command of 0x000F0110_12345678 is interpreted as SIZE = 0, BE = 0xF, ADDR = 0x0110, and DATA = 0x12345678. SIZE = 0 means single access is used to write the data 0x12345678 to the register at address 0x0110.

## 8.7.4. Burst Access

When SIZE is 1 or greater, there are two or more data items, and data is written to registers using burst access. With burst access, SIZE+1 data items (up to 256 items) are written to one or multiple registers.

DATA stores the first 32 bits of data. The second and later data items are stored contiguously, with two items in each of the 64-bit segments that follow. The lower 32 bits store the first item in each pair, and the upper 32 bits store the second item. Because 3D commands are 64-bit aligned, the upper 32 bits of the final 64-bit segment are ignored when writing an odd number of data items.

**Figure 8-6. Data Storage Order When Using Burst Access**




The BE value is applied to all data to be written. That is, if BE is 0, no data is written to any register.

## 8.7.4.1. Writing Single Registers

When SEQ is 0, the access mode is single-register writing. In this mode multiple data items are written in succession to a single register.

Data is written in succession only to the register specified in ADDR.

Example:

The 3D command:
0x004F0080_11111111
0x33333333_22222222
0x55555555_44444444
is interpreted as SIZE = 4, BE = 0xF, ADDR = 0x0080, DATA = 0x11111111, and SEQ = 0. Because SIZE = 4 and SEQ = 0, data is written using burst access in single-register mode, with all five data items (0x11111111, 0x22222222, 0x33333333, 0x44444444, 0x55555555) being written one after another to the register at address 0x0080. The next 3D command to run is stored in the 64-bit segment following 0x55555555_44444444.

## 8.7.4.2. Sequential Register Writing

When SEQ is 1, the access mode is sequential register writing. In this mode, each data item is written only once to multiple sequential registers.

One data item is written per register to the registers coming sequentially after the address specified in ADDR (with the address incremented by one each time).

Example:

The 3D command:
0x805F0280_11111111
0x33333333_22222222
0x55555555_44444444
0x77777777_66666666 is interpreted as SIZE = 5, BE = 0xF, ADDR = 0x0280, DATA

= 0x11111111, and SEQ = 1. Because SIZE = 5 and SEQ = 1, data is written using burst access in sequential register mode, with one data item written to each of the six registers starting from address 0x0280. 0x11111111 is written to the register at address 0x0280, 0x22222222 to the register at 0x0281, 0x33333333 to the register at 0x0282, 0x44444444 to the register at 0x0283, 0x55555555 to the register at 0x0284, and 0x66666666 to the register at 0x0285. Given the SIZE value of 5, 0x77777777 is padding and is ignored. The next 3D command to run is stored in the 64-bit segment following 0x77777777_66666666.

## 8.7.5. 3D Command Execution Cost

Each 3D command that writes a value to a PICA register requires one clock cycle to write once to one register (except for certain registers), for either single or burst access.

The rasterization module outputs a busy signal for one cycle each time it processes a 3D command. Consequently, each 3D command sent to modules in the process flow, starting with the rasterization module (rasterization module, texture unit, fragment lighting, texture combiner, and per-fragment operation module), will take two cycles to process.

Commands to clear the texture cache (bit [16:16] of register 0x0080) or the post-vertex cache (register 0x0231) both require one clock cycle. However, although processing each command takes one cycle, 3D commands are commands to the texture unit, so only one command can be input per two cycles.

Commands to flush the framebuffer cache (bit [0:0] of register 0x0111) require around 100 cycles, and commands to clear the early depth buffer (bit [0:0] of register 0x0063) require around 1000 cycles.

In addition, entering a 3D command into the triangle setup, rasterization, texture unit, fragment lighting, texture combiner, or per-fragment operation modules when there is still fragment data in the module (when the module is processing a fragment) flushes the pipeline once per module. Consequently, 3D commands entered immediately after a render command require a pipeline flush for each of these modules.

The following table shows the register ranges allocated for these modules.

**Table 8-12. Register Ranges Allocated to Modules**

| Module | Register Range |
|---|---|
| Triangle setup | 0x0040 to 0x005F |
| Rasterization module | 0x0060 to 0x006F |
| Texture unit | 0x0080 to 0x00BF |
| Fragment lighting | 0x0140 to 0x01DF |
| Texture combiners | 0x00C0 to 0x00FF |
| Per-fragment operation module | 0x0100 to 0x013F |

**Note:** This includes addresses that do not actually have a register.

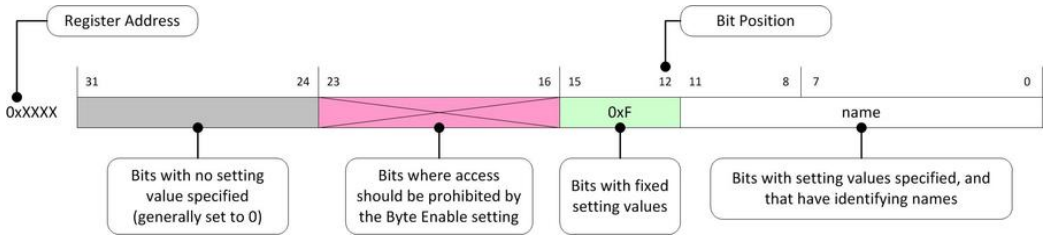
# 8.8. PICA Register Information

This section describes addressing, how to set values, and value formats for various PICA registers. Based on this information, you can change the setting values for a feature by searching for the locations in the 3D Command Buffer where commands write to the relevant registers, and overwriting those locations.

**Warning:** When setting a memory address in a register, you must call the `nn::gx::GetPhysicalAddr()` function to convert the virtual memory address to a physical address.

The general format for register bit layouts is as follows.

**Figure 8-7. Register Bit Layout Legend**




## 8.8.1. Vertex Shader Setting Registers (0x02B0 - 0x02DF and Others)

This section describes the registers used for settings involved with vertex shaders, such as vertex shader starting addresses, vertex attributes, and the settings of floating-point constant registers.

### 8.8.1.1. Floating-Point Constant Registers (0x02C0, 0x02C1 - 0x02C8)

Vertex shaders have 96 floating-point constant registers (expressed in shader assembly code as `c0` through `c95`), each of which is comprised of the four components `xyzw`. These can be set either by using the shader assembly `def` instruction to define a constant, or by using a uniform to define a constant. When using shader assembly code, the value is set in the GPU's internal format as a 24-bit floating-point value (the lowest 16 bits are the significand, followed by 7 bits for the exponent and 1 bit for the sign). When using a uniform, the value is set as a 32-bit floating-point value (expressed as an IEEE 754 format single-precision floating point number), which is then automatically converted in the GPU to 24 bits.

#### Index Specifier (0x02C0)

Register `0x02C0` specifies which data input mode is used to write data to which floating-point constant register.

**Figure 8-8. Bit Layout for the Floating-Point Constant Register Index Specifier (0x02C0)**

INDEX specifies the index of a floating-point constant register. Setting this to 0x00 specifies register c0, 0x0A specifies register c10, and 0x5F specifies register c95. At the same time, setting MODE to 1 sets the data input mode to 32-bit floating-point, and setting MODE to 0 sets the data input mode to 24-bit floating-point.

The data to write to the four floating-point constant register components (x, y, z, and w) is written to one of the registers 0x02C1 through 0x02C8. The results are identical no matter which of these registers is written to. For an extreme example, writing all the data to one register produces the same result as writing the data to registers in reverse order starting from 0x02C8.

To set floating-point constant register values, first write the index and mode to 0x02C0, and then write the data to registers 0x02C1 through 0x02C8.

### 32-Bit Floating-Point Input Mode

In 32-bit floating-point input mode, four 32-bit data items are written to registers in the 0x02C1 to 0x02C8 range to set the value of one floating-point constant register. These four components are written in the order w, z, y, and then x.

After these four, 32-bit data items are written, the index is automatically incremented by 1, so that the next floating-point constant register to be set will be the one whose index is next after the specified index. In other words, if the index in register 0x02C0 is set to 0x0A, the first four data items written to the 0x02C1 to 0x02C8 registers are then written to register c10, and the next four data items are written to c11.

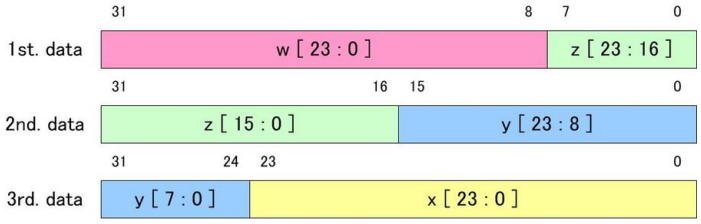
Example:

> Writing 0x80000023 to register 0x02C0 is interpreted as MODE = 1, INDEX =35, preparing the system to write to floating-point constant register c35 in 32-bit floating-point input mode. Then, if you write 0x40800000 to register 0x02C1, 0x40400000 to 0x02C2, 0x40000000 to 0x02C3, and 0x3F800000 to 0x02C4, and repeat this combination again (a total of 8 writes), registers c35.xyzw and c36.xyzw are both set to { 1.0f, 2.0f, 3.0f, 4.0f }.
>
> These settings could be expressed with 3D commands like the following.
> 0x000F02C0_80000023
> 0x803F02C1_40800000 0x40000000_40400000 0x00000000_3F800000
> 0x803F02C1_40800000 0x40000000_40400000 0x00000000_3F800000

### 24-Bit Floating-Point Input Mode

In 24-bit floating-point input mode, four 24-bit data items are packed into three 32-bit data segments, and then written to registers in the 0x02C1 to 0x02C8 range to set the value of one floating-point constant register. The data is packed into the 32-bit segments in the component order w, z, y, and then x. The following figure shows how four 24-bit data items are packed into three 32-bit data items. For more information about converting 32-bit floating-point values to 24-bit floating-point values, see 8.9.1. Conversion to a 24-Bit Floating-Point Number.

**Figure 8-9. Data Layout in 24-bit Floating-Point Input Mode**




After these three 32-bit data items are written, the index is automatically incremented by 1, so that the next floating-point constant register to be set will be the one whose index is next after the specified index. In other words, if the index in register `0x02C0` is set to `0x0A`, the first three data items written to the `0x02C1` to `0x02C8` registers are then written to register `c10`, and the next three data items are written to `c11`.

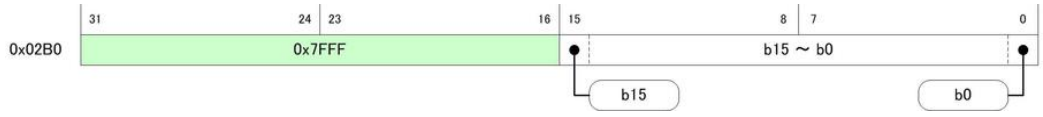
Example:

> Writing `0x80000023` to register `0x02C0` is interpreted as `MODE = 0`, `INDEX =35`, preparing the system to write to floating-point constant register `c35` in 24-bit floating-point input mode. If you then write `0x40800000` to register `0x02C1`, `0x40400000` to `0x02C2`, `0x40000000` to `0x02C3`, and `0x3F800000` to `0x02C4`, and repeat this combination again (a total of 6 write operations), registers `c35.xyzw` and `c36.xyzw` are both set to { `1.0f, 2.0f, 3.0f, 4.0f` }.
>
> These settings could be expressed with 3D commands like the following.
> `0x000F02C0_00000023`
> `0x802F02C1_41000040 0x003F0000_80004000`
> `0x802F02C1_41000040 0x003F0000_80004000`

## 8.8.1.2. Boolean Register (0x02B0)

Vertex shaders have 16 Boolean registers (expressed in shader assembly code as registers `b0` through `b15`). These can be set either by using the shader assembly `def` instruction to define a constant, or by using a uniform to define a constant.

**Figure 8-10. Bit Layout of Boolean Register (0x02B0)**




The bits [15:0] in register `0x02B0` correspond one-to-one with the vertex shader Boolean registers. Bit [0] corresponds to register `b0`, bit [1] with `b1`, and so on through bit [15] and register `b15`. Writing a value of 1 represents `TRUE`, and a value of `0` represents `FALSE`.

Note that the 15$^{th}$ Boolean register (`b15`) is reserved by the geometry shader when the geometry shader is used.

## 8.8.1.3. Integer Registers (0x02B1 - 0x02B4)

Vertex shaders have four floating-point constant registers (expressed in shader assembly code as `i0` through `i3`), each of which is comprised of the four components `xyzw`. These can be set either by using the shader assembly `defi` instruction to define a constant, or by using a uniform

to define a constant.

**Figure 8-11. Bit Layout of Integer Registers (0x02B1 - 0x02B4)**

Register 0x02B1 corresponds to i0, 0x02B2 to i1, 0x02B3 to i2, and 0x02B4 to i3. Each register stores the three components x, y, and z in 8 bits each, starting from the lowest bit of the register. Setting negative numbers to y and z is expressed with two's complement.

### 8.8.1.4. Program Code Setting Registers (0x02BF, 0x02CB – 0x02D3, 0x02D5 – 0x02DD)

There are multiple registers used to load the program code executed by vertex shaders. Specifically, there are registers that specify the addresses to which to load the programs and registers for writing program data.

**Figure 8-12. Bit Layout of Program Code Loading Registers (0x02CB – 0x02D3)**

In the ADDR portion of register 0x02CB, set the address to which to load the vertex shader program code. In registers 0x02CC through 0x02D3, write the data for the program code to load.

After you set the program code loading address in register 0x02CB, write the data to any of the registers 0x02CC through 0x02D3. Each instruction in a vertex shader program is 32 bits, so writing one data item corresponds to writing one instruction, and the loading address is incremented by one after every write. The results are the same no matter which of these registers is written to.

After you have updated the program code, you must write 1 to any of the bits in register 0x02BF to notify the GPU that program updating is complete.

In addition to loading the program code as described above, you must also load the swizzle pattern data. The following figure shows the registers that set swizzle patterns.

**Figure 8-13. Bit Layout of Swizzle Pattern Loading Registers (0x02D5 – 0x02DD)**

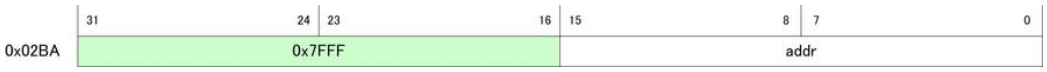
In the ADDR portion of register 0x02D5, set the address to which to load the swizzle pattern. In registers 0x02D6 through 0x02DD, write the swizzle pattern data to load.

After you set the swizzle pattern loading address in register 0x02D5, write the data to any of the registers 0x02D6 through 0x02DD. The loading address is incremented by one after every write. The results are the same no matter which of these registers is written to.

## 8.8.1.5. Starting Address Setting Register (0x02BA)

The address of the `main` label defined in the shader assembly code is the vertex shader starting address.

**Figure 8-14. Bit Layout of Starting Address Setting Register (0x02BA)**

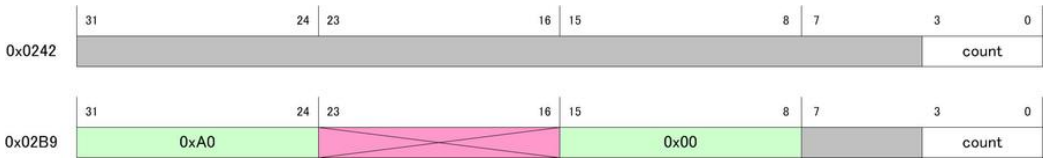



Set the vertex shader starting address in bits [15:0] of register `0x02BA`.

## 8.8.1.6. Attribute Input Count Setting Registers (0x0242, 0x02B9)

There are multiple registers used to set the number of vertex attributes to input to a vertex shader. The same value is set in each of these registers.

**Figure 8-15. Bit Layout of Vertex Attribute Input Count Setting Registers (0x0242, 0x02B9)**




Set the value of `count` to the number of vertex attributes to input minus 1. Up to 12 vertex attributes can be input when the vertex buffer is used (when vertex data is loaded using a load array). When the vertex buffer is not used (when vertex data is loaded via the command buffer), up to 16 vertex attributes can be input.

## 8.8.1.7. Input Register Mapping Setting Registers (0x02BB, 0x02BC)

The following shows the registers used to set the mapping between the vertex attribute data to input to the vertex shader and the input registers.

**Figure 8-16. Bit Layout of Input Register Mapping Setting Registers (0x02BB, 0x02BC)**

In `attrib_0` through `attrib_15`, set index numbers designating which input registers store the vertex attribute data to input to the vertex shader. (Register `v0` is indicated by a value of `0x0`, `v1` by `0x1` and so on, with `v15` indicated by `0xF`.) The order of the vertex attributes to input to the vertex shader does not correspond to the order specified by the *`index`* parameter of the `glBindAttribLocation()` function. It corresponds instead with the internal vertex attribute numbers described in 8.8.1.9. Vertex Attribute Array Setting Registers (0x0200 – 0x0227).

## 8.8.1.8. Fixed Vertex Attribute Value Setting Registers (0x0232 – 0x0235)

The fixed vertex attribute values set by functions such as `glVertexAttrib4f` are converted to 24-bit floating point numbers and transferred to the GPU. The fixed vertex attribute values are transferred to the GPU using the settings of the following registers.

**Figure 8-17. Bit Layout of Fixed Vertex Attribute Value Setting Registers (0x0232 – 0x0235)**

First, the internal vertex attribute number of these fixed vertex attribute values is written to bits [3:0] of register `0x0232`. The fixed vertex attribute values are then converted to 24-bit floating point numbers and written, in order, to the three, 32-bit data segments in registers `0x0233`, `0x0234`, and `0x0235`. This 24-bit floating-point data written to the three, 32-bit segments is created in the same way as the data presented in 24-Bit Floating-Point Input Mode.

When you switch a vertex array from enabled to disabled or vice versa by using the registers described in 8.8.1.9. Vertex Attribute Array Setting Registers (0x0200 – 0x0227), it invalidates any fixed vertex attribute values that had previously been set. You must reset these fixed vertex attribute values. Also note that GPU specifications do not allow you to use all the vertex attributes as fixed vertex attributes without also using one or more vertex arrays. Always use at least one vertex array when you use fixed vertex attributes.

## 8.8.1.9. Vertex Attribute Array Setting Registers (0x0200 – 0x0227)

There are multiple registers used to set vertex attribute arrays when using vertex buffers. The commands for setting these registers are generated on validation of the state flag `NN_GX_STATE_FRAMEBUFFER`.

**Figure 8-18. Bit Layout of Vertex Attribute Array Setting Registers (0x0200 and Others)**

The settings to these registers include the base address, the internal vertex attribute type, the fixed vertex attribute mask, the total number of vertex attributes, each load array's byte offset, information about the load array elements, the number of load array elements, the load array size in bytes, and the index array's offset.

### Base Address

A value consisting of the physical address divided by 16 is set as the base address in `ARRAY_BASE_ADDR` in register `0x0200`. All the vertex array addresses and the address of the vertex index array are set as the base address plus an offset. If the range of addresses for the vertex arrays and index array has been fixed in advance, the base address does not need to be reset for each combination of vertex arrays.

### Internal Vertex Attribute Type

Internal vertex attributes are vertex attribute numbers determined internally by the GPU so that it can load vertex arrays. These numbers are different from the vertex attribute numbers specified by the *`index`* parameter of the `glEnableVertexAttribArray()` function (hereafter called the *GL vertex attribute numbers*), but there is a one-to-one correspondence between internal vertex attributes and GL vertex attribute numbers.

Vertex arrays enabled by the `glEnableVertexAttribArray()` function are assigned internal vertex attributes as values starting at 0 and increasing sequentially with no numbering gaps, but GL vertex attribute number 0 does not necessarily correlate to internal vertex attribute 0. For instance, when the vertex arrays with GL vertex attribute numbers 0 and 3 have been enabled, they may be assigned internal vertex attributes 0 and 1, or they may be assigned the other way around as 1 and 0. The method by which internal vertex attribute values are assigned depends on the driver implementation, and is subject to change.

In the current implementation, the vertex array addresses are sorted in ascending order, and assigned internal vertex attributes, starting from 0 for the array with the leading GL vertex attribute.

The vertex attribute data is input to the vertex shader in the internal vertex attribute order.

The ARRAY_TYPE*n* (*n* = 0 to 11) portion of registers 0x0201 – 0x0202 sets the type of the array having the $(n + 1)^{th}$ internal vertex attribute. The value set to the register as the internal vertex attribute type is determined by the combination of the *size* and *type* parameters specified in the glVertexAttribPointer() function. The following table shows how these combinations correspond with the values set to the register.

**Table 8-13. size and type Combinations and Values Set to the Register**

| size | type | Setting Value |
|---|---|---|
| 1 | GL_BYTE | 0x0 |
| 1 | GL_UNSIGNED_BYTE | 0x1 |
| 1 | GL_SHORT | 0x2 |
| 1 | GL_FLOAT | 0x3 |
| 2 | GL_BYTE | 0x4 |
| 2 | GL_UNSIGNED_BYTE | 0x5 |
| 2 | GL_SHORT | 0x6 |
| 2 | GL_FLOAT | 0x7 |
| 3 | GL_BYTE | 0x8 |
| 3 | GL_UNSIGNED_BYTE | 0x9 |
| 3 | GL_SHORT | 0xA |
| 3 | GL_FLOAT | 0xB |
| 4 | GL_BYTE | 0xC |
| 4 | GL_UNSIGNED_BYTE | 0xD |
| 4 | GL_SHORT | 0xE |
| 4 | GL_FLOAT | 0xF |

### Fixed Vertex Attribute Mask

The number of enabled vertex attributes is defined by a #pragma bind_symbol statement in the vertex shader assembly code, but if those enabled vertex attributes have disabled vertex arrays (vertex attributes on which the glDisableVertexAttribArray() function has been called or on which the glEnableVertexAttribArray() function has not been called), fixed vertex attributes are used for them.

Much like vertex arrays, fixed vertex attributes are assigned internal vertex attributes. Fixed vertex attributes are assigned internal vertex attributes, starting from the last number assigned to a vertex array and increasing sequentially, with no numbering gaps.

A mask of the assigned internal vertex attributes is set by the CONST_ATTRIB_MASK portion of register 0x0202. This mask corresponds to internal vertex attributes 0 through 11, starting from its least-significant bit and ascending in order. Its bits are set to 1 for those internal vertex attributes that are assigned to fixed vertex attributes.

When you enable or disable a vertex array using these register settings, it invalidates any fixed vertex attribute values that had previously been set. You must reset these fixed vertex attribute values. Also note that GPU specifications do not allow you to use all the vertex attributes as fixed vertex attributes without also using one or more vertex arrays. Always use at least one vertex array when you use fixed vertex attributes.

### Number of Vertex Attributes

Set the ARRAY_NUM portion of register 0x0202 to the total number of fixed vertex attributes, plus vertex attributes that use vertex arrays, minus 1.

### Load Arrays

To load vertex attribute data, the GPU manages vertex attribute arrays in units of data arrays. These data arrays are called load arrays, and the GPU loads data from 12 load arrays.

These 12 load arrays each have 12 elements. Each load array element contains either vertex array data that makes up the load array, or 4-byte units of padding. In general, if you have defined your vertex data as an array of structures holding multiple vertex attributes (this is called an interleaved array), that single interleaved array corresponds to a single load array. Conversely, if you define your vertex data as an array holding one vertex attribute (an independent array), that single vertex attribute corresponds to a single load array.

Load arrays must be used in ascending order from the first load array (load array 0). For example, you cannot use a combination like load array 1 and load array 4 that does not start from 0 and is not consecutive.

The actual address for a vertex attribute array is the value derived by adding the address allocated by the `glBufferData()` function to the offset specified by the `glVertexAttribPointer()` function's *ptr* parameter. When setting the registers, this actual address is set as the result of this formula: [base address × 16 + the byte offset to the load array].

The ARRAY*n*_OFFSET (*n* = 0 to 11) portions of registers (0x0203 + *n* × 3) set the byte offsets to each $(n + 1)^{th}$ load array. GPU performance improves when there are fewer load arrays used, so the driver is configured to load all the data in as few load arrays as possible.

Set the ARRAYn_ELEMi (*n* = 0 to 11, *i* = 0 to 11) portions of registers (0x0204 + *n* × 3) through (0x0205 + *n* × 3) to the $(i + 1)^{th}$ element of the $(n + 1)^{th}$ load array, starting from the first element of the first array. Set each element to either the internal vertex attribute used or padding. The values set to the register correspond to the elements as follows.

**Table 8-14. Values Set to Registers and Corresponding Elements**

| Setting Value | Element |
|---|---|
| 0x0 | Internal vertex attribute 0 |
| 0x1 | Internal vertex attribute 1 |
| 0x2 | Internal vertex attribute 2 |
| 0x3 | Internal vertex attribute 3 |
| 0x4 | Internal vertex attribute 4 |
| 0x5 | Internal vertex attribute 5 |
| 0x6 | Internal vertex attribute 6 |
| 0x7 | Internal vertex attribute 7 |
| 0x8 | Internal vertex attribute 8 |
| 0x9 | Internal vertex attribute 9 |
| 0xA | Internal vertex attribute 10 |
| 0xB | Internal vertex attribute 11 |
| 0xC | 4-byte padding |
| 0xD | 8-byte padding |

| | |
|---|---|
| 0xE | 12-byte padding |
| 0xF | 16-byte padding |

For example, when `0x0` is set in `ARRAY0_ELEM0` in register `0x0204`, the first element of the first load array is set to internal vertex attribute 0. The data type of the data placed at the start of the load array's data structure is the data type of internal vertex attribute 0, which is set in `ARRAY_TYPE0` in register `0x0201`.

Set the `ARRAY`*n*`_STRIDE` portion of registers (`0x0205 + `*n*` × 3`) to the number of bytes per vertex in the (*n* + 1)$^{th}$ load array. Load arrays having elements of multiple different types sometimes automatically include padding. `ARRAY`*n*`_STRIDE` must set the number of bytes in the array which include padding. If the value set there contradicts the total size of the elements in the load array, operation is undefined. Set the `ARRAY`*n*`_ATTRIB_NUM` (*n* = 0 to 11) portion of registers (`0x0205 + `*n*` × 3`) to the number of attributes in the (*n* + 1)$^{th}$ load array. One load array can sometimes include multiple vertex attribute arrays, for example when multiple vertex attribute arrays are laid out as interleaved arrays. The value set as the number of attributes in the (*n* + 1)$^{th}$ load array is not the same as the number of vertex attribute arrays included in that load array. If the load array's number of attributes is set to 0, that load array is not used.

Set the `INDEX_ARRAY_OFFSET` portion of register `0x0227` to the byte offset to the index array. Set the `INDEX_ARRAY_TYPE` portion of register `0x0227` to the vertex index type. Set this type value to 1 when the *type* parameter to the `glDrawElements()` function is `GL_UNSIGNED_SHORT`, and 0 when type is `GL_UNSIGNED_BYTE`. Always set to 1 when using the `glDrawArrays()` function.

Example 1: Interleaved Array

**Code 8-15. Interleaved Array Example (Structure)**

```
struct vertex_t
{
    float position[3];
    float color[4];
    float texcoord[2];
} vertex[NUM_VERTEX];
```

When vertex data consists of this structure, the vertex array settings are configured as follows.

**Code 8-16. Interleaved Array Example (Vertex Array Settings)**

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(struct vertex_t), 0);
glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE, sizeof(struct vertex_t), 12);
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, sizeof(struct vertex_t), 28);
```

In this case, the three GL vertex attributes 0, 1, and 2 comprise the elements of one load array. If the only used vertex attributes are these three and one fixed vertex attribute, the vertex attributes 0, 1, and 2 correspond to internal vertex attributes 0, 1, and 2, and the fixed vertex attribute corresponds to internal vertex attribute 3. The related register settings are as follows.

`0x0201: 0x000007FB` // Internal vertex attribute types are `0:FLOAT_VEC3`, `1:FLOAT_VEC4`, `2:FLOAT_VEC2`.
`0x0202: 0x30080000` // Four vertex attributes in total, with internal vertex attribute 3 as a fixed vertex attribute.
`0x0203: 0x00000000` // Only one load array is used, so the base address is set to the actual address.
`0x0204: 0x00000210` // The elements of load array 0 are the internal vertex attributes 0,

1, and 2.
`0x0205: 0x30240000` // There are `float × 9 = 36` bytes per each vertex in load array 0, and the number of elements is 3.
`0x0206 – 0x0226: 0x00000000` // Other load arrays not used.

Example 2: Independent Array

**Code 8-17. Independent Array Example (Structure)**

```
#define NUM_VERTEX (3)
struct attribute0_t
{
    float position[3];
} attirbute0[NUM_VERTEX];
struct attribute1_t
{
    float color[4];
} attribute1[NUM_VERTEX];
struct attribute2_t
{
    float tex[2];
} attribute2[NUM_VERTEX];
```

When vertex data consists of this structure, the vertex array settings are configured as follows. The vertex buffer is a single shared object and assumes that the data is laid out in order.

**Code 8-18. Independent Array Example (Vertex Array Settings)**

```
glBindBuffer(GL_ARRAY_BUFFER, 1);
glBufferData(GL_ARRAY_BUFFER,

sizeof(attribute0)+sizeof(attribute1)+sizeof(attribute2),0,GL_STATIC_DRAW);
glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(attribute0), attribute0);
glBufferSubData(GL_ARRAY_BUFFER,
        sizeof(attribute0), sizeof(attribute1), attribute1);
glBufferSubData(GL_ARRAY_BUFFER,
        sizeof(attribute0)+sizeof(attribute1), sizeof(attribute2), attribute2);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE, 0,
        (GLvoid*)(sizeof(attribute0)));
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 0,
        (GLvoid*)(sizeof(attribute0)+sizeof(attribute1)));
```

In this case, the three GL vertex attributes 0, 1, and 2 each comprise elements of separate load arrays, corresponding to internal vertex attributes 0, 1, and 2, respectively. The related register settings are as follows.

`0x0201: 0x000007FB` // Internal vertex attribute types are `0:FLOAT_VEC3`, `1:FLOAT_VEC4`, `2:FLOAT_VEC2`.
`0x0202: 0x20000000` // Three vertex attributes in total, with no fixed vertex attributes.
`0x0203: 0x00000000` // Load array 0 is located at the start.
`0x0204: 0x00000000` // The element of load array 0 is one element of internal vertex attribute 0.
`0x0205: 0x100C0000` // There are `float × 3 = 12` bytes per each vertex in load array 0, and the number of elements is 1.
`0x0206: 0x00000024` // The offset to load array 1 is `sizeof(attribute0)`.
`0x0207: 0x00000001` // The element of load array 1 is one element of internal vertex attribute 1.
`0x0208: 0x10100000` // There are `float × 4 = 16` bytes per each vertex in load array 1, and the number of elements is 1.
`0x0209: 0x00000054` // The offset to load array 2 is `sizeof(attribute0) + sizeof(attribute1)`.

`0x020A: 0x00000002` // The element of load array 2 is one element of internal vertex attribute 2.
`0x020B: 0x10080000` // There are `float × 2 = 8` bytes per each vertex in load array 2, and the number of elements is 1.
`0x020C – 0x0226: 0x00000000` // Other load arrays not used.

### Load Array Padding Elements and Automatic Padding

If the ARRAY*n*_ELEM*i* (*n* = 0 to 11, *i* = 0 to 11) portion of registers (`0x0204 + n × 3`) through (`0x0205 + n × 3`) is set to any values `0xC` through `0xF`, that element is padding. Padding is used when the load arrays include regions not used as vertex attributes.

For example, we can create vertex data that uses structures like the following.

**Code 8-19. Example of a Structure Using Padding**

```
struct vertex_t
{
    float position[3];
    float color[4];
    float texcoord[2];
} vertex[NUM_VERTEX];
```

Consider a situation in which `texcoord` shown in this structure is not used as a vertex attribute. The byte size per vertex is `float × 9`, but the last `float × 2` bytes is not used. As a result, internal vertex attributes are specified in the first and second elements of the load array corresponding to this vertex data, but the third element specifies `0xD` (8 bytes of padding).

Padding cannot be specified in the first element. Behavior is undefined when it is specified. Set the byte offset of the load array adjusted so that there is no padding in the first element.

When one load array has elements containing vertex attributes of multiple different data types (`GL_FLOAT`, `GL_SHORT`, `GL_BYTE`, or `GL_UNSIGNED_BYTE`), up to 4 bytes of padding is sometimes automatically inserted, even when no padding is specified by the load array elements. Elements in a load array have several possible sizes. Each individual element comprising a load array might be either a 4-byte type (an internal vertex attribute of type `GL_FLOAT`, or padding), a 2-byte type (an internal vertex attribute of type `GL_SHORT`), or a 1-byte type (an internal vertex attribute of type `GL_BYTE` or `GL_UNSIGNED_BYTE`). Padding is automatically inserted for each element of the load array, so that elements align to their own particular sizes. Padding is also automatically inserted at the end of each instance of vertex data, so that the vertex data is aligned to the size of the biggest data type contained by any element in the array.

For example, you can create vertex data that uses structures like the following.

**Code 8-20. Example of Automatic Padding**

```
struct vertex_t
{
    GLfloat position[3];
    GLubyte color[3];
    GLfloat texcoord[2];
    GLubyte param;
} vertex[NUM_VERTEX];
```

Assume that the load array is composed of the elements of four vertex attributes of `position`, `color`, `texcoord`, and `param`. As shown above, `color` is 3 bytes, but the data coming immediately after it in the array (`texcoord`, of type `GLfloat`) is aligned to 4 bytes. This means that one byte of padding is automatically inserted right after `color`.

Because the element with the largest size in the load array is of type `GLfloat`, padding is also automatically inserted after each instance of vertex data to ensure 4-byte alignment of the vertex data. This means that three bytes of padding are automatically inserted right after `param`.

#### Load Array Settings and Performance

The performance of loading vertex data depends on the size and number of the load arrays you use, and on other factors such as the type of elements that they contain.

The GPU accesses memory in units of one load array at a time, but there is no cache and it uses the same resources to load multiple load arrays from different addresses as it does to load multiple load arrays from the same address.

When you want to load the same vertex array into multiple vertex shader input registers, one approach is to load multiple load arrays from the same address. However, an alternative approach is to make several copies of that vertex array to create an interleaved array, and load the interleaved array. This second approach entails a larger data size than the first approach, but it may have better runtime performance.

Even when loading the same size of vertex data in either case, loading it via a single load array has better performance than loading it via multiple load arrays. This difference in performance is less pronounced when the vertex indices are optimized to be consecutive, and is also affected by other modules that access device memory. For example, even without accounting for access conflicts in device memory, it takes approximately 1.3 to 2 times as long for two load arrays to each load three `GLfloat` vertex data values than for a single load array to load all six values. However, both these cases will have the same performance if the vertex arrays are placed in VRAM.

#### Load Array Limit

A maximum of 11 load arrays can be used when rendering with the `glDrawElements()` function (start rendering by writing to register `0x022F`). When using 12 vertex attributes to render with the `glDrawElements()` function, at least 1 vertex attribute must be static, and at least 2 must be interleaved arrays. Use no more than 11 load arrays. If 12 load arrays are used to start rendering with the `glDrawElements()` function, the GPU may freeze. The GPU can also freeze in other cases where the load array setting is inappropriate.

If the GPU freezes due to an illegal load array setting, the value obtained by the `nnqxGetCmdlistParameteri()` function in `NN_GX_CMDLIST_HW_STATE` enters a state where 1 is set at the eighth bit only.

## 8.8.1.10. Output Register Use Count Setting Registers (0x004F, 0x024A, 0x0251, 0x025E)

There are multiple registers used to set the number of vertex attributes output from a vertex shader. Set the same value in most of these registers.

**Figure 8-19. Bit Layout of Output Register Use Count Setting Registers (0x004F, 0x024A, 0x0251, 0x025E)**

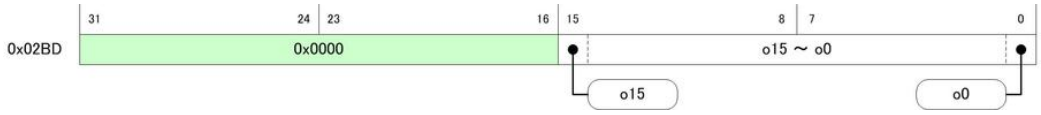
Set the unchanged raw number of output registers to use in *count1* (register 0x004F), and set the number of used output registers minus 1 in *count2* (registers 0x024A, 0x0251, and 0x025E). Note that *count1* has a different value and bit width from the others.

The number of output registers is defined by a #pragma output_map statement in the vertex shader assembly code. Because this is the number of output registers actually used, this number is 1 when multiple vertex attributes are packed into one output register.

## 8.8.1.11. Output Register Mask Setting Register (0x02BD)

The register that sets which output registers are written by vertex shaders contains a mask of 16 bits, corresponding to the 16 output registers.

**Figure 8-20. Bit Layout of Output Register Mask Setting Register (0x02BD)**




The bits [15:0] in register 0x02BD correspond one-to-one with the output registers. Bit [0] is register o0, bit [1] is register o1, and so on, up to bit [15] for register o15. In this mask, set the bits corresponding to the output registers defined by the #pragma output_map statement to 1. Set the bits corresponding to output registers not so defined to 0.

## 8.8.1.12. Output Register Attribute Setting Registers (0x0050 – 0x0056, 0x0064)

There are seven output registers available to output vertex attributes from vertex shaders. There are multiple registers used to set the attributes output in each component of these output registers. Set the attributes in order of output register, starting from the lowest-numbered used output register.

**Figure 8-21. Bit Layout of Output Register Attribute Setting Registers (0x0050 – 0x0056, 0x0064)**

The names in the bit layout correspond to the following output attribute settings. The following table also shows the number of bits and possible values of each name.

**Table 8-15. Names and Their Settings (Output Attribute Setting Registers)**

| Name | Bits | Description |
|---|---|---|
| attrib_x<br>attrib_y<br>attrib_z<br>attrib_w | 5 | These are the vertex attributes set to the x, y, z, and w components of the output registers, starting from the top.<br>0x00: Vertex coordinate x component.<br>0x01: Vertex coordinate y component.<br>0x02: Vertex coordinate z component.<br>0x03: Vertex coordinate w component.<br>0x04: Quaternion x component.<br>0x05: Quaternion y component.<br>0x06: Quaternion z component.<br>0x07: Quaternion w component.<br>0x08: Vertex color red component.<br>0x09: Vertex color green component.<br>0x0A: Vertex color blue component.<br>0x0B: Vertex color alpha component.<br>0x0C: Texture coordinate 0, u component.<br>0x0D: Texture coordinate 0, v component.<br>0x0E: Texture coordinate 1, u component.<br>0x0F: Texture coordinate 1, v component.<br>0x10: Texture coordinate 0, w component.<br>0x12: View vector x component.<br>0x13: View vector y component.<br>0x14: View vector z component.<br>0x16: Texture coordinate 2, u component.<br>0x17: Texture coordinate 2, v component.<br>0x1F: Disabled |
| texcoord | 1 | Sets whether to include texture coordinates in the vertex attributes output from vertex shaders.<br>0x0: Do not output texture coordinates<br>0x1: Output texture coordinates |

The following example describes the register settings when the vertex shaders are defined as follows.

**Code 8-21. Example of Setting Output Attributes**

```
#pragma output_map(position, o0)
#pragma output_map(color, o1)
#pragma output_map(texture0, o2.xy)
#pragma output_map(texture0w, o2.z)
#pragma output_map(texture1, o3.xy)
```

Registers are set as follows.

```
0x0050: 0x03020100
0x0051: 0x0B0A0908
0x0052: 0x1F100D0C // The w component is disabled.
0x0053: 0x1F1F0F0E // The zw components are disabled.
0x0054: 0x1F1F1F1F // The 5th attribute is disabled.
0x0055: 0x1F1F1F1F // The 6th attribute is disabled.
0x0056: 0x1F1F1F1F // The 7th attribute is disabled.
0x0064: 0x00000001 // Output texture coordinates.
```

## 8.8.1.13. Output Attribute Clock Control Register (0x006F)

Depending on what kinds of vertex attributes are output from vertex shaders, sometimes you must

change the setting of the clock control register.

**Figure 8-22. Bit Layout of Output Attribute Clock Control Register (0x006F)**

The names in the bit layout correspond to the following output attribute clock control settings. The following table also shows the number of bits and possible values of each name.

**Table 8-16. Names and Their Settings (Output Attribute Clock Control Register)**

| Name | Bits | Description |
|---|---|---|
| vectorZ | 1 | Set to 1 to output the z component of vertex coordinates and 0 to not output it. |
| vertexColor | 1 | Set to 1 to output the vertex color and 0 to not output it. |
| texture0 | 1 | Set to 1 to output texture coordinate 0 and 0 to not output it. |
| texture1 | 1 | Set to 1 to output texture coordinate 1 and 0 to not output it. |
| texture2 | 1 | Set to 2 to output texture coordinate 1 and 0 to not output it. |
| texture0w | 1 | Set to 1 to output the w coordinate of texture coordinate 0 and 0 to not output it. |
| viewVector | 1 | Set to 1 to output view vectors and quaternions and 0 to not output them. |

These bits control power supply to the modules involved with the corresponding vertex attributes. Set the bits corresponding to unused vertex attributes to 0 to reduce power consumption.

## 8.8.2. Texture Address Setting Registers (0x0085 – 0x008A, 0x0095, 0x009D)

There are multiple registers used to set texture data addresses. For texture unit 0 there are 2D texture and cube map texture settings, and for texture units 1 and 2 there are 2D texture settings.

This section describes just the settings for addresses of texture data bound to the various targets. You can change the layout of texture data using this information. To change the resolution, filter mode, number of mipmap levels, or other characteristics of textures, see 8.8.6. Texture Setting Registers (0x0080, 0x0083, 0x008B, 0x00A8 – 0x00B7 and Others).

**Figure 8-23. Bit Layout of Texture Address Setting Registers (0x0085 and Others)**

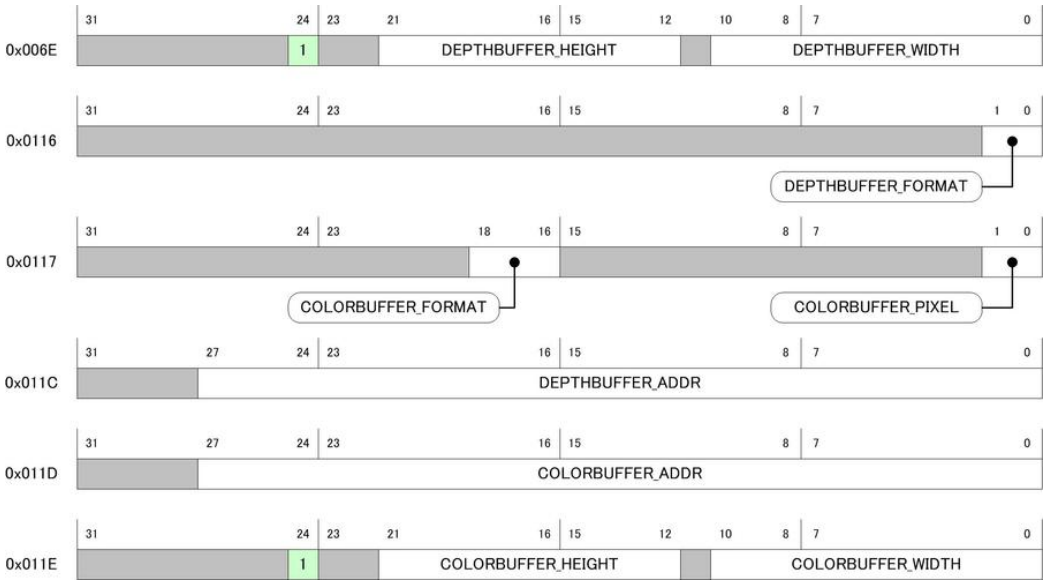
Texture addresses are all 8-byte addresses. (An 8-byte address is the value resulting when the physical address is divided by 8.) The highest 6 bits of the 28-bit combined texture address of all six faces of a cube map are shared with bits [27:22] of register `0x0085`.

> **Warning:** Texture addresses must be 128-byte aligned. If texture addresses are not correctly aligned, various phenomena may occur, such as the GPU may hang or rendered results may be corrupted.

## 8.8.3. Render Buffer Setting Registers (0x006E, 0x0116, 0x0117, 0x011C – 0x011E)

Among the render buffer-related settings, there are multiple registers for setting the color and depth buffers. The commands for setting these registers are generated on validation of the state flag `NN_GX_STATE_FRAMEBUFFER`.

**Figure 8-24. Bit Layout of Render Buffer Setting Registers (0x006E, 0x0116, 0x0117, 0x011C – 0x011E)**




The names in the bit layout correspond to the following render buffer settings. The following table also shows the number of bits and possible values of each name.

**Table 8-17. Names and Their Settings (Render Buffer Setting Registers)**

| Name | Bits | Description |
|---|---|---|
| COLORBUFFER_FORMAT | 3 | Sets the color buffer format.<br>0x0 : GL_RGBA8_OES or GL_GAS_DMP<br>0x2 : GL_RGB5_A1<br>0x3 : GL_RGB565<br>0x4 : GL_RGBA4 |
| COLORBUFFER_PIXEL | 2 | Sets the pixel size of the color buffer format.<br>0x0: 16-bit<br>0x2: 32-bit |
| COLORBUFFER_WIDTH | 11 | Sets the color buffer width in pixels. |
| COLORBUFFER_HEIGHT | 10 | Sets the color buffer height in pixels, minus 1. |
| COLORBUFFER_ADDR | 28 | Sets the color buffer address as an 8-byte address (a physical address divided by 8). |
| DEPTHBUFFER_FORMAT | 2 | Sets the depth buffer format.<br>0x0 : GL_DEPTH_COMPONENT16<br>0x2 : GL_DEPTH_COMPONENT24_OES<br>0x3 : GL_DEPTH24_STENCIL8_EXT |
| DEPTHBUFFER_WIDTH | 11 | Sets the depth buffer width in pixels. |
| DEPTHBUFFER_HEIGHT | 10 | Sets the depth buffer height in pixels, minus 1. |
| DEPTHBUFFER_ADDR | 28 | Sets the depth buffer address as an 8-byte address (a physical address divided by 8). |

Always specify 0xF in byte-enable for the setting command for register 0x011E (COLORBUFFER_WIDTH and COLORBUFFER_HEIGHT).

## 8.8.4. Texture Combiner Setting Registers (0x00C0 – 0x00C4 and Others)

The register settings that set the dmp_TexEnv[*i*] texture combiner reserved uniforms are split among multiple registers. These registers start at a different address (comb_top) for each differently numbered combiner.

**Table 8-18. Combiner Numbers and the Starting Addresses of Their Registers**

| Combiner Number | Starting Address (comb_top) |
|---|---|
| 0 | 0x00C0 |
| 1 | 0x00C8 |
| 2 | 0x00D0 |
| 3 | 0x00D8 |
| 4 | 0x00F0 |
| 5 | 0x00F8 |

**Figure 8-25. Bit Layout of Texture Combiner Setting Registers (0x00C0 – 0x00C4 and Others)**

The names in the bit layout correspond to the following reserved uniforms. The following table also shows the number of bits and possible values of each name.

**Table 8-19. Names and Their Reserved Uniforms (Texture Combiner Setting Registers)**

| Name | Bits | Description |
|---|---|---|
| srcRgb0<br>srcRgb1<br>srcRgb2 | 4 | From the top, the first through third elements of the values set in dmp_TexEnv[*i*].srcRgb.<br>0x0 : GL_PRIMARY_COLOR<br>0x1 : GL_FRAGMENT_PRIMARY_COLOR_DMP<br>0x2 : GL_FRAGMENT_SECONDARY_COLOR_DMP<br>0x3 : GL_TEXTURE0<br>0x4 : GL_TEXTURE1<br>0x5 : GL_TEXTURE2<br>0x6 : GL_TEXTURE3<br>0xE : GL_CONSTANT<br>0xF : GL_PREVIOUS<br>0xD : GL_PREVIOUS_BUFFER_DMP |
| srcAlpha0<br>srcAlpha1 | 4 | From the top, the first through third elements of the values set in dmp_TexEnv[*i*].srcAlpha. |

| | | |
|---|---|---|
| srcAlpha2 | | The settings are the same as for srcRgb0 through srcRgb2. |
| operandRgb0<br>operandRgb1<br>operandRgb2 | 4 | From the top, the first through third elements of the values set in dmp_TexEnv[*i*].operandRgb.<br>0x0 : GL_SRC_COLOR<br>0x1 : GL_ONE_MINUS_SRC_COLOR<br>0x2 : GL_SRC_ALPHA<br>0x3 : GL_ONE_MINUS_SRC_ALPHA<br>0x4 : GL_SRC_R_DMP<br>0x5 : GL_ONE_MINUS_SRC_R_DMP<br>0x8 : GL_SRC_G_DMP<br>0x9 : GL_ONE_MINUS_SRC_G_DMP<br>0xC : GL_SRC_B_DMP<br>0xD : GL_ONE_MINUS_SRC_B_DMP |
| operandAlpha0<br>operandAlpha1<br>operandAlpha2 | 3 | From the top, the first through third elements of the values set in dmp_TexEnv[*i*].operandAlpha.<br>0x0 : GL_SRC_ALPHA<br>0x1 : GL_ONE_MINUS_SRC_ALPHA<br>0x2 : GL_SRC_R_DMP<br>0x3 : GL_ONE_MINUS_SRC_R_DMP<br>0x4 : GL_SRC_G_DMP<br>0x5 : GL_ONE_MINUS_SRC_G_DMP<br>0x6 : GL_SRC_B_DMP<br>0x7 : GL_ONE_MINUS_SRC_B_DMP |
| combineRgb | 4 | The values set in dmp_TexEnv[*i*].combineRgb.<br>0x0 : GL_REPLACE<br>0x1 : GL_MODULATE<br>0x2 : GL_ADD<br>0x3 : GL_ADD_SIGNED<br>0x4 : GL_INTERPOLATE<br>0x5 : GL_SUBTRACT<br>0x6 : GL_DOT3_RGB<br>0x7 : GL_DOT3_RGBA<br>0x8 : GL_MULT_ADD_DMP<br>0x9 : GL_ADD_MULT_DMP |
| combineAlpha | 4 | The values set in dmp_TexEnv[*i*].combineAlpha.<br>These are the same as the combineRgb values, excluding GL_DOT3_RGB. |
| constRgba0<br>constRgba1<br>constRgba2<br>constRgba3 | 8 | From the top, the first through third elements of the values set in dmp_TexEnv[*i*].constRgba.<br>Set using unsigned 8-bit integers, mapping the values 0.0 through 1.0 to the numbers 0 through 255.<br>For more information about how these values are converted, see 8.9.16. Conversion From a Floating-Point Number (Between 0 and 1) to an 8-Bit Unsigned Integer. |
| scaleRgb<br>scaleAlpha | 2 | The values set in dmp_TexEnv[*i*].scaleRgb and dmp_TexEnv[*i*].scaleAlpha.<br>0x0 : 1.0<br>0x1 : 2.0<br>0x2 : 4.0 |

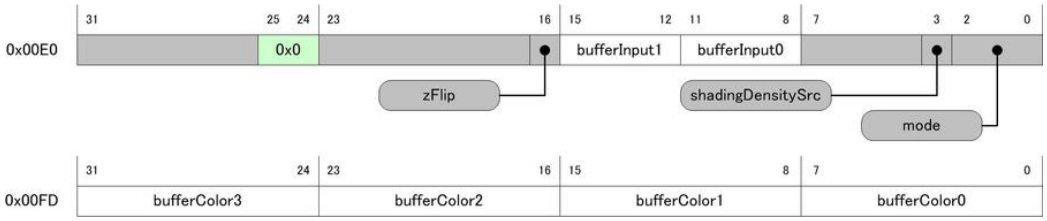
dmp_TexEnv[*i*].srcRgb and dmp_TexEnv[*i*].srcAlpha settings have the following restrictions:

- When *i* is 0, GL_PREVIOUS and GL_PREVIOUS_BUFFER_DMP cannot be configured for the three elements of dmp_TexEnv[*i*].srcRgb and dmp_TexEnv[*i*].srcAlpha.
- When *i* is not 0, at least one of the three elements in each of dmp_TexEnv[*i*].srcRgb and dmp_TexEnv[*i*].srcAlpha must be set to one of GL_CONSTANT, GL_PREVIOUS, or GL_PREVIOUS_BUFFER_DMP.

### 8.8.4.1. Combiner Buffer Setting Registers (0x00E0, 0x00FD)

The reserved uniforms with names beginning with `dmp_TexEnv[i].` are also used for combiner buffer settings. The registers used to set values in combiner buffer reserved uniforms have the following layout. Note that the other bits in register `0x00E0` are used for other settings (such as gas settings).

**Figure 8-26. Bit Layout of Combiner Buffer Setting Registers (0x00E0, 0x00FD)**




The names in the bit layout correspond to the following reserved uniforms. The following table also shows the number of bits and possible values of each name.

**Table 8-20. Names and Their Reserved Uniforms (Combiner Buffer Setting Registers)**

| Name | Bits | Description |
|---|---|---|
| bufferColor0<br>bufferColor1<br>bufferColor2<br>bufferColor3 | 4 | From the top, the first through fourth elements of the values set in `dmp_TexEnv[0].bufferColor.`<br>Set using unsigned 8-bit integers, mapping the values `0.0` through `1.0` to the numbers 0 through 255.<br>For more information about how these values are converted, see 8.9.16. Conversion From a Floating-Point Number (Between 0 and 1) to an 8-Bit Unsigned Integer. |
| bufferInput0<br>bufferInput1 | 1 *<br>4 | From the top, the first through second elements of the values set in `dmp_TexEnv[i].bufferInput`. *i* is 1 through 4, with the least significant bit corresponding to a setting of 1.<br>`0x0` : `GL_PREVIOUS_BUFFER_DMP`<br>`0x1` : `GL_PREVIOUS` |

## 8.8.5. Fragment Lighting Setting Registers (0x008F and Others)

This section describes registers involved with reserved uniforms used in fragment lighting. These are reserved uniforms that include `dmp_FragmentLighting`, `dmp_FragmentMaterial`, `dmp_FragmentLightSource[i]`, or `dmp_LightEnv` in their names.

### 8.8.5.1. Lighting Enable/Disable Control Registers (0x008F, 0x01C2, 0x01C6, 0x01D9)

The registers corresponding to the reserved uniforms that enable and disable lighting have the following layout.

**Figure 8-27. Bit Layout of Lighting Enable/Disable Control Registers (0x008F, 0x01C2, and Others)**

The names in the bit layout correspond to the following reserved uniforms. The following table also shows the number of bits and possible values of each name.

**Table 8-21. Names and Their Reserved Uniforms (Lighting Enable/Disable Control Registers)**

| Name | Bits | Description |
|---|---|---|
| enabled0 | 1 | The value set in `dmp_FragmentLighting.enabled`.<br>`0x0`: `GL_FALSE`<br>`0x1`: `GL_TRUE` |
| enabled1 | 1 | The value set in `dmp_FragmentLighting.enabled`.<br>`0x0`: `GL_TRUE`<br>`0x1`: `GL_FALSE`<br>**Note:** Pay close attention to these values. |
| src_num | 3 | Set to the [*number of enabled light sources* – 1]. Set to `0` when all light sources are disabled.<br>The number of enabled light sources is the number of light sources for which `dmp_FragmentLightSource[`*i*`].enabled` is set to `GL_TRUE`. |
| id1 through id8 | 3 | Enabled light source IDs are assigned starting from `id1`.<br>If light sources 0, 1, 3, and 5 are enabled, this register is set to `0x00005310`. (There is no restriction on order of the light sources. The setting could also be `0x00003150`). When all light sources are enabled, this register is set to `0x76543210`. When all light sources are disabled, this register is set to `0x00000000`.<br>When the same light source is specified multiple times, the lighting result for that light source will be applied multiple times. When light source 0 is specified multiple times, the primary color's global ambient setting will also be applied multiple times in addition to the lighting result. |

## 8.8.5.2. Global Ambient Setting Register (0x01C0)

The global ambient settings are configured by setting the values for the RGB components in individual bits of the `0x01C0` register. The values to set are calculated using the equation below, clamped to the range [`0.0,1.0`], and then mapped to the unsigned 8-bit integers 0 through 255. For more information about how these values are converted, see 8.9.16. Conversion From a Floating-Point Number (Between 0 and 1) to an 8-Bit Unsigned Integer. If light source 0 is not enabled, this register's setting is ignored and a value of `0` is used for the primary color's global ambient term.

dmp_FragmentMaterial.emission + dmp_FragmentMaterial.ambient × dmp_FragmentLighing.ambient

Example:

When the emissive light and ambient light for a material are (`0.8, 0.8, 0.8, 0.6`) and (`0.2, 0.2, 0.2, 0.4`), respectively, and the global ambient light is (`1.0, 1.0, 1.0, 1.0`), the value to set for the components is obtained by converting the value calculated below. The final converted value is 143 (`0x8F`).

$$(0.8 \times 0.6) + (0.2 \times 0.4) \times (1.0 \times 1.0) = 0.56$$

The bit layout for the setting registers is as follows. Although 10 bits are provided for each component, the calculated results are stored in the lower 8 bits and `0` is stored in the upper 2 bits. (The figure shows in which 8 bits the values are actually set.) Behavior is undefined when the upper 2 bits are not set to `0`.

**Figure 8-28. Bit Layout of Global Ambient Setting Register (0x01C0)**

If lighting is enabled (`dmp_FragmentLighting.enabled` is `GL_TRUE`) and all light sources are disabled (`dmp_FragmentLightSource[i].enabled` are `GL_FALSE`), only the global ambient color is applied to the primary color.

Register `0x01C2` sets the number of enabled light sources, and the value of its bits [2:0] is the [*number of light sources* - 1]. Consequently, enabling lighting also enables one light source, even if that is not what you intended. In such cases, the driver generates commands that set black (`0.0, 0.0, 0.0, 0.0`) in all color components of light source 0 (in other words, registers `0x0140` through `0x0143` are set to 0). The driver also ensures that light source 0 is the enabled light source by generating commands that set light source 0 to be the first light source enabled (commands that set bits [2:0] of register `0x01D9` equal to `0x0`). In addition, it generates commands that set `dmp_LightEnv.config` equal to `GL_LIGHT_ENV_LAYER_CONFIG0_DMP` (commands that set bits [7:4] of register `0x01C3` equal to `0x0`).

## 8.8.5.3. Light Source Setting Registers (0x0140 – 0x01BF, 0x01C4)

All individual light settings are based on light source numbers. There are multiple registers for these settings. Their starting address `light_top` is based on this formula: [`light_top` = `0x0140` + light source number × `0x10`]. For example, to set the colors of light sources 0 and 3 in `dmp_FragmentLightSource[0].specular0` and `dmp_FragmentLightSource[3].specular0`, the corresponding registers are `0x140` and `0x170` respectively.

### Light Source Color Setting Registers (0x0140 – 0x0143 and Others)

Light source colors are configured by setting the RGB component values for specular light 0 (`LightSpecular0`), specular light 1 (`LightSpecular1`), diffuse light (`LightDiffuse`), and ambient light (`LightAmbient`) in the individual bits of the registers starting from the light_top addresses. The values to set are calculated using the equation below, clamped to the range [`0.0,1.0`], and then mapped to the unsigned 8-bit integers 0 through 255. For more information about how these values are converted, see 8.9.16. Conversion From a Floating-Point Number (Between 0 and 1) to an 8-Bit Unsigned Integer.

`LightSpecular0 = dmp_FragmentMaterial.specular0 × dmp_FragmentLightSource[i].specular0`

When `dmp_LightEnv.lutEnabledRefl` is `GL_FALSE`:

```
LightSpecular1 = dmp_FragmentMaterial.specular1 ×
dmp_FragmentLightSource[i].specular1
```

When `dmp_LightEnv.lutEnabledRefl` is `GL_TRUE`:

```
LightSpecular1 = dmp_FragmentLightSource[i].specular1
```

```
LightDiffuse = dmp_FragmentMaterial.diffuse ×
dmp_FragmentLightSource[i].diffuse
```

```
LightAmbient = dmp_FragmentMaterial.ambient ×
dmp_FragmentLightSource[i].ambient
```

The bit layout for the setting registers is as follows.

**Figure 8-29. Bit Layout of Light Source Color Setting Registers (0x0140 – 0x0143 and Others)**

### Light Source Position Setting Registers (0x0144, 0x0145, 0x0149 and Others)

The `dmp_FragmentLightSource[i].position` reserved uniform sets light source position coordinates. The `xyz` components of these coordinates are converted to 16-bit floating-point values before being set to registers. For more information about how these values are converted, see 8.9.2. Converting to a 16-Bit Floating-Point Number. For the `w` component, set whether the light source is point or directional in bit [`0:0`] of the (`light_top + 9`) register. Note that the bit is set to `1` if the `w` component value is `0.0`, and set to `0` otherwise. Also note that the other bits in this register are used for separate settings.

The bit layout for the setting registers is as follows.

**Figure 8-30. Bit Layout of Light Source Position Setting Registers (0x0144, 0x0145, 0x0149 and Others)**

### Spotlight Direction Setting Registers (0x0146, 0x0147 and Others)

The `dmp_FragmentLightSource[i].spotDirection` reserved uniform sets the coordinates of the spotlight direction. Before these `xyz` coordinates are set to registers, their signs are flipped, and they are converted to signed 13-bit fixed-point numbers with 11 fractional bits (using two's complement to represent negative values). For more information about how these values are converted, see 8.9.9. Conversion to a 13-Bit Signed Fixed-Point Number With 11 Fractional Bits.

The bit layout for the setting registers is as follows.

**Figure 8-31. Bit Layout of Spotlight Direction Setting Registers (0x0146, 0x0147 and Others)**

## Distance Attenuation Setting Registers (0x014A, 0x014B and Others)

The distance attenuation bias and scale values set by the distance attenuation reserved uniforms `dmp_FragmentLightSource[i].distanceAttenuationBias` and `dmp_FragmentLightSource[i].distanceAttenuationScale` are set to registers after being converted to 20-bit floating point numbers. For more information about how these values are converted, see 8.9.4. Converting to a 20-Bit Floating-Point Number.

The bit layout for the setting registers is as follows.

**Figure 8-32. Bit Layout of Distance Attenuation Setting Registers (0x014A, 0x014B and Others)**

## Other Setting Registers (0x01C4, 0x0149 and Others)

The registers that correspond to other settings for each light source have the following layout.

**Figure 8-33. Bit Layout of Other Setting Registers (0x01C4, 0x0149 and Others)**

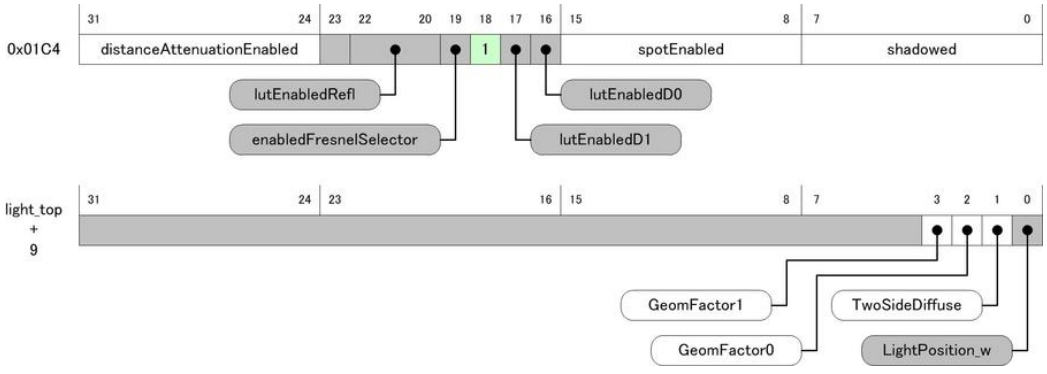



**Table 8-22. Reserved Uniforms and Registers for Other Light Source Settings**

| Name/Register | Bits | Description |
|---|---|---|
| shadowed<br>0x01C4 bit [ x : x ]<br>(x is the light source number.) | 1 | The value set in dmp_FragmentLightSource[*i*].shadowed.<br>0x0: GL_TRUE<br>0x1: GL_FALSE<br>**Note:** Pay close attention to these values. |
| spotEnabled<br>0x01C4 bit [ 8 + x : 8 + x ]<br>(x is the light source number.) | 1 | The value set in dmp_FragmentLightSource[*i*].spotEnabled.<br>0x0: GL_TRUE<br>0x1: GL_FALSE<br>**Note:** Pay close attention to these values. |
| distanceAttenuationEnabled<br>0x01C4 bit [ 24 + x : 24 + x ]<br>(x is the light source number.) | 1 | The value set in dmp_FragmentLightSource[*i*].distanceAttenuationEnabled<br>0x0: GL_TRUE<br>0x1: GL_FALSE<br>**Note:** Pay close attention to these values. |
| twoSideDiffuse<br>(light_top + 9), bit [1:1] | 1 | The value set in dmp_FragmentLightSource[*i*].twoSideDiffuse.<br>0x0: GL_FALSE<br>0x1: GL_TRUE |
| geomFactor0<br>(light_top + 9), bit [2:2] | 1 | The value set in dmp_FragmentLightSource[*i*].geomFactor0.<br>0x0: GL_FALSE<br>0x1: GL_TRUE |
| geomFactor1<br>(light_top + 9), bit [3:3] | 1 | The value set in dmp_FragmentLightSource[*i*].geomFactor1.<br>0x0: GL_FALSE<br>0x1: GL_TRUE |

## 8.8.5.4. Lookup Table Setting Registers (0x01C5, 0x01C8 – 0x01CF)

Fragment lighting lookup tables specified by the dmp_FragmentMaterial.sampler{RR, RG, RB, D0, D1, FR} and dmp_FragmentLightSource[*i*].sampler{SP, DA} reserved uniforms are set using 256 data items and the same number of delta values.

The bit layout for the setting registers is as follows.

**Figure 8-34. Bit Layout of Lookup Table Setting Registers (0x01C5, 0x01C8 – 0x01CF)**

In register 0x01C5, set Ref_Table to the lookup table you are targeting and Ref_Index to the index at which to start setting the table. An index of 0 indicates the first data item, and 255 indicates the last data item. The values set in Ref_Table correspond to the following lookup tables.

**Table 8-23. Ref_Table Values and Lookup Tables**

| Ref_Table | Targeted Lookup Table |
|---|---|
| 0x0 | Distribution factor 0 (D0) |
| 0x1 | Distribution factor 1 (D1) |
| 0x3 | Fresnel factors (FR) |
| 0x4 | Blue component of reflection (RB) |

| | |
|---|---|
| 0x5 | Green component of reflection (RG) |
| 0x6 | Red component of reflection (RR) |
| 0x8 + *i* | Spotlight (SP), where *i* is the light source number |
| 0x10 + *i* | Light's distance attenuation (DA), where *i* is the light source number |

To set registers `0x01C8` through `0x01CF`, write the values derived from combining the $i^{th}$ data item and $(i + 256)^{th}$ delta value in the lookup table loaded by the `glTexImage1D()` function. Set `Ref_Value` to the data item converted to an unsigned 12-bit fixed-point number with 12 fractional bits, and set `Ref_Difference` to the delta value converted to a signed 12-bit fixed-point number with 11 fractional bits. (The fractional portion is an absolute value, so negative numbers are not expressed using two's complement.) For more information about how these values are converted, see 8.9.13. Conversion to a 12-Bit Unsigned Fixed-Point Number With 12 Fractional Bits and 8.9.6 Conversion to a 12-Bit Signed Fixed-Point Number With 11 Fractional Bits.

After setting the targeted lookup table and index in register `0x01C5`, write the sets of converted values to any register in the range `0x01C8` – `0x01CF`. The results are the same no matter which of these registers is written to, and the index is incremented by one for each data item that is written.

## 8.8.5.5. Lookup Table Argument Range Setting Register (0x01D0)

The register for setting the lookup table argument ranges has the following layout.

**Figure 8-35. Bit Layout of Lookup Table Argument Range Setting Register (0x01D0)**

**Table 8-24. Reserved Uniforms and Register for Setting Lookup Table Argument Ranges**

| Name/Register | Bits | Description |
|---|---|---|
| `absLutInputD0`<br>0x01D0 bit [ 1 : 1 ] | 1 | The value set in `dmp_LightEnv.absLutInputD0`.<br>0x0 : GL_TRUE<br>0x1 : GL_FALSE<br>**Note:** Pay close attention to these values. |
| `absLutInputD1`<br>0x01D0 bit [ 5 : 5 ] | 1 | The value set in `dmp_LightEnv.absLutInputD1`.<br>0x0 : GL_TRUE<br>0x1 : GL_FALSE<br>**Note:** Pay close attention to these values. |
| `absLutInputSP`<br>0x01D0 bit [ 9 : 9 ] | 1 | The value set in `dmp_LightEnv.absLutInputSP`.<br>0x0 : GL_TRUE<br>0x1 : GL_FALSE<br>**Note:** Pay close attention to these values. |
| `absLutInputFR`<br>0x01D0 bit [ 13 : 13 ] | 1 | The value set in `dmp_LightEnv.absLutInputFR`.<br>0x0 : GL_TRUE<br>0x1 : GL_FALSE<br>**Note:** Pay close attention to these values. |
| `absLutInputRB` | | The value set in `dmp_LightEnv.absLutInputRB`.<br>0x0 : GL_TRUE |

| | | |
|---|---|---|
| 0x01D0 bit [ 17 : 17 ] | 1 | 0x1 : GL_FALSE<br>**Note:** Pay close attention to these values. |
| absLutInputRG<br>0x01D0 bit [ 21 : 21 ] | 1 | The value set in dmp_LightEnv.absLutInputRG.<br>0x0 : GL_TRUE<br>0x1 : GL_FALSE<br>**Note:** Pay close attention to these values. |
| absLutInputRR<br>0x01D0 bit [ 25 : 25 ] | 1 | The value set in dmp_LightEnv.absLutInputRR.<br>0x0 : GL_TRUE<br>0x1 : GL_FALSE<br>**Note:** Pay close attention to these values. |

## 8.8.5.6. Lookup Table Input Values Setting Register (0x01D1)

The register for setting the lookup table input values has the following layout.

**Figure 8-36. Bit Layout of Lookup Table Input Value Setting Register (0x01D1)**



**Table 8-25. Reserved Uniforms and Register for Setting Lookup Table Input Values**

| Name/Register | Bits | Description |
|---|---|---|
| lutInputD0<br>0x01D1 bit [ 2 : 0 ] | 3 | The value set in dmp_LightEnv.lutInputD0.<br>0x0 : GL_LIGHT_ENV_NH_DMP<br>0x1 : GL_LIGHT_ENV_VH_DMP<br>0x2 : GL_LIGHT_ENV_NV_DMP<br>0x3 : GL_LIGHT_ENV_LN_DMP<br>0x4 : GL_LIGHT_ENV_SP_DMP<br>0x5 : GL_LIGHT_ENV_CP_DMP |
| lutInputD1<br>0x01D1 bit [ 6 : 4 ] | 3 | The value set in dmp_LightEnv.lutInputD1.<br>The description is the same as lutInputD0. |
| lutInputSP<br>0x01D1 bit [ 10 : 8 ] | 3 | The value set in dmp_LightEnv.lutInputSP.<br>The description is the same as lutInputD0. |
| lutInputFR<br>0x01D1 bit [ 14 : 12 ] | 3 | The value set in dmp_LightEnv.lutInputFR.<br>0x0: GL_LIGHT_ENV_NH_DMP<br>0x1 : GL_LIGHT_ENV_VH_DMP<br>0x2 : GL_LIGHT_ENV_NV_DMP<br>0x3 : GL_LIGHT_ENV_LN_DMP |
| lutInputRB<br>0x01D1 bit [ 18 : 16 ] | 3 | The value set in dmp_LightEnv.lutInputRB.<br>The description is the same as lutInputFR. |
| lutInputRG<br>0x01D1 bit [ 22 : 20 ] | 3 | The value set in dmp_LightEnv.lutInputRG.<br>The description is the same as lutInputFR. |
| lutInputRR<br>0x01D1 bit [ 26 : 24 ] | 3 | The value set in dmp_LightEnv.lutInputRR.<br>The description is the same as lutInputFR. |

## 8.8.5.7. Lookup Table Output Value Scale Setting Register (0x01D2)

The register for setting the scale values to apply to the lookup table output values has the following layout.

**Figure 8-37. Bit Layout of Scale Value Setting Register (0x01D2) to Apply to the Lookup Table Output Values**

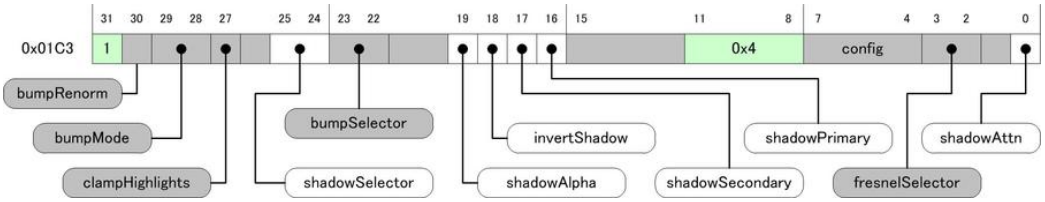
**Table 8-26. Reserved Uniforms and Register for Setting Lookup Table Output Value Scaling**

| Name/Register | Bits | Description |
|---|---|---|
| `lutScaleD0`<br>0x01D2 bit [ 2 : 0 ] | 3 | The value set in `dmp_LightEnv.lutScaleD0`.<br>0x0 : 1.0<br>0x1 : 2.0<br>0x2 : 4.0<br>0x3 : 8.0<br>0x6 : 0.25<br>0x7 : 0.5 |
| `lutScaleD1`<br>0x01D2 bit [ 6 : 4 ] | 3 | The value set in `dmp_LightEnv.lutScaleD1`.<br>The description is the same as `lutScaleD0`. |
| `lutScaleSP`<br>0x01D2 bit [ 10 : 8 ] | 3 | The value set in `dmp_LightEnv.lutScaleSP`.<br>The description is the same as `lutScaleD0`. |
| `lutScaleFR`<br>0x01D2 bit [ 14 : 12 ] | 3 | The value set in `dmp_LightEnv.lutScaleFR`.<br>The description is the same as `lutScaleD0`. |
| `lutScaleRB`<br>0x01D2 bit [ 18 : 16 ] | 3 | The value set in `dmp_LightEnv.lutScaleRB`.<br>The description is the same as `lutScaleD0`. |
| `lutScaleRG`<br>0x01D2 bit [ 22 : 20 ] | 3 | The value set in `dmp_LightEnv.lutScaleRG`.<br>The description is the same as `lutScaleD0`. |
| `lutScaleRR`<br>0x01D2 bit [ 26 : 24 ] | 3 | The value set in `dmp_LightEnv.lutScaleRR`.<br>The description is the same as `lutScaleD0`. |

## 8.8.5.8. Shadow Attenuation Setting Register (0x01C3)

The register for setting the shadow attenuation has the following layout. Note that the other bits in this register are used for separate settings.

**Figure 8-38. Bit Layout of Shadow Attenuation Setting Register (0x01C3)**




The names in the bit layout correspond to the following reserved uniforms. The following table also shows the number of bits and possible values of each name.

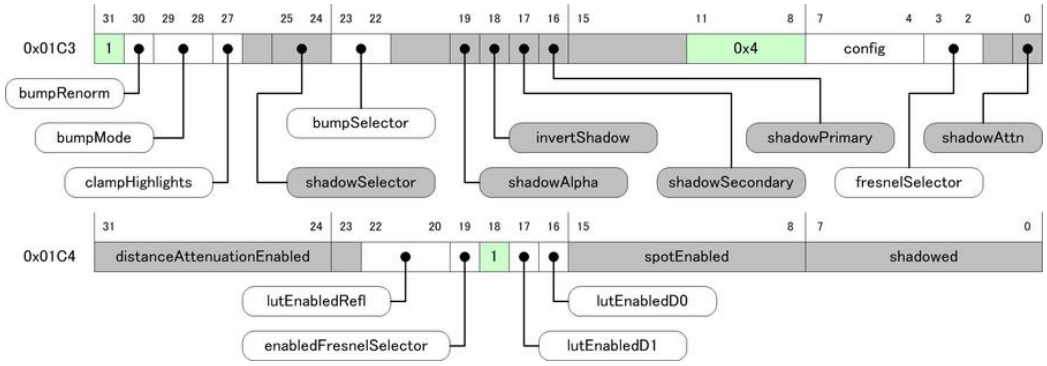
**Table 8-27. Names and Their Reserved Uniforms (Shadow Attenuation Setting Register)**

| Name | Bits | Description |
|---|---|---|
| shadowSelector | 2 | The value set in dmp_LightEnv.shadowSelector.<br>0x0 : GL_TEXTURE0<br>0x1 : GL_TEXTURE1<br>0x2 : GL_TEXTURE2<br>0x3 : GL_TEXTURE3 |
| shadowPrimary | 1 | The value set in dmp_LightEnv.shadowPrimary.<br>0x0: GL_FALSE<br>0x1: GL_TRUE |
| shadowSecondary | 1 | The value set in dmp_LightEnv.shadowSecondary.<br>0x0: GL_FALSE<br>0x1: GL_TRUE |
| invertShadow | 1 | The value set in dmp_LightEnv.invertShadow.<br>0x0: GL_FALSE<br>0x1: GL_TRUE |
| shadowAlpha | 1 | The value set in dmp_LightEnv.shadowAlpha.<br>0x0: GL_FALSE<br>0x1: GL_TRUE |
| shadowAttn | 1 | Set to 1 when any of dmp_LightEnv.shadowPrimary, dmp_LightEnv.shadowSecondary, or dmp_LightEnv.shadowAlpha is GL_TRUE. Set to 0 when all of these are GL_FALSE. |

## 8.8.5.9. Other Setting Registers (0x01C3, 0x01C4)

The registers for other fragment lighting settings have the following layout. Note that the other bits in this register are used for separate settings.

**Figure 8-39. Bit Layout of Other Setting Registers (0x01C3, 0x01C4)**




The names in the bit layout correspond to the following reserved uniforms. The following table also shows the number of bits and possible values of each name.

**Table 8-28. Names and Their Reserved Uniforms (Other Setting Registers)**

| Name | Bits | Description |
|---|---|---|
| config | 4 | The value set in dmp_LightEnv.config.<br>0x0 : GL_LIGHT_ENV_LAYER_CONFIG0_DMP<br>0x1 : GL_LIGHT_ENV_LAYER_CONFIG1_DMP<br>K : GL_LIGHT_ENV_LAYER_CONFIG2_DMP<br>0x3 : GL_LIGHT_ENV_LAYER_CONFIG3_DMP<br>0x4 : GL_LIGHT_ENV_LAYER_CONFIG4_DMP<br>0x5 : GL_LIGHT_ENV_LAYER_CONFIG5_DMP<br>0x6 : GL_LIGHT_ENV_LAYER_CONFIG6_DMP |

| | | |
|---|---|---|
| | | 0x8 : GL_LIGHT_ENV_LAYER_CONFIG7_DMP |
| fresnelSelector | 2 | The value set in dmp_LightEnv.fresnelSelector.<br>0x0 : GL_LIGHT_ENV_NO_FRESNEL_DMP<br>0x1 : GL_LIGHT_ENV_PRI_ALPHA_FRESNEL_DMP<br>0x2 : GL_LIGHT_ENV_SEC_ALPHA_FRESNEL_DMP<br>0x3 : GL_LIGHT_ENV_PRI_SEC_ALPHA_FRESNEL_DMP |
| enabledFresnelSelector | 1 | Changes depending on the value set in dmp_LightEnv.fresnelSelector.<br>0x0: Any setting other than GL_LIGHT_ENV_NO_FRESNEL_DMP<br>0x1: GL_LIGHT_ENV_NO_FRESNEL_DMP |
| bumpSelector | 2 | The value set in dmp_LightEnv.bumpSelector.<br>0x0: GL_TEXTURE0<br>0x1: GL_TEXTURE1<br>0x2: GL_TEXTURE2<br>0x3: GL_TEXTURE3 |
| bumpMode | 2 | The value set in dmp_LightEnv.bumpMode.<br>0x0 : GL_LIGHT_ENV_BUMP_NOT_USED_DMP<br>0x1 : GL_LIGHT_ENV_BUMP_AS_BUMP_DMP<br>0x2 : GL_LIGHT_ENV_BUMP_AS_TANG_DMP |
| bumpRenorm | 1 | The value set in dmp_LightEnv.bumpRenorm.<br>0x0: GL_TRUE, or dmp_LightEnv.bumpMode is GL_LIGHT_ENV_BUMP_NOT_USED_DMP. |
| clampHighlights | 1 | The value set in dmp_LightEnv.clampHighlights.<br>0x0: GL_FALSE<br>0x1: GL_TRUE |
| lutEnabledD0 | 1 | The value set in dmp_LightEnv.lutEnabledD0.<br>0x0: GL_TRUE<br>0x1: GL_FALSE<br>**Note:** Pay close attention to these values. |
| lutEnabledD1 | 1 | The value set in dmp_LightEnv.lutEnabledD1.<br>0x0: GL_TRUE<br>0x1: GL_FALSE<br>**Note:** Pay close attention to these values. |
| lutEnabledRefl | 3 | The value set in dmp_LightEnv.lutEnabledRefl.<br>0x0: GL_TRUE<br>0x1: GL_FALSE<br>**Note:** Pay close attention to these values. |

The value of config (bits [7:4] of register 0x01C3) changes the number of cycles used for pixel operations. Because the config setting has an effect even when lighting is disabled, when lighting is turned off, make sure to change it to a setting that causes pixel operations to use one cycle. When lighting is disabled, the driver sets bits [7:4] of register 0x01C3 equal to 0x0. In addition, when the setting value is (8:GL_LIGHT_ENVLAYER_CONFIG7_DMP), distance attenuation can no longer be used. If the distance attenuation setting remains enabled, an illegal value will be applied to distance attenuation, so set bits [31:24] of register 0x01C4 to 0xFF to disable distance attenuation

## 8.8.6. Texture Setting Registers (0x0080, 0x0083, 0x008B, 0x00A8 – 0x00B7 and Others)

This section describes registers involved with reserved uniforms that make texture settings (uniforms whose names include dmp_Texture[*i*]). The commands for setting these registers are generated on validation of the state flag NN_GX_STATE_TEXTURE. Also see 8.8.2. Texture Address

## 8.8.6.1. Shadow Texture Setting Register (0x008B)

The register for setting shadow textures has the following layout.

**Figure 8-40. Bit Layout of Shadow Texture Setting Register (0x008B)**

The names in the bit layout correspond to the following reserved uniforms. The following table also shows the number of bits and possible values of each name.

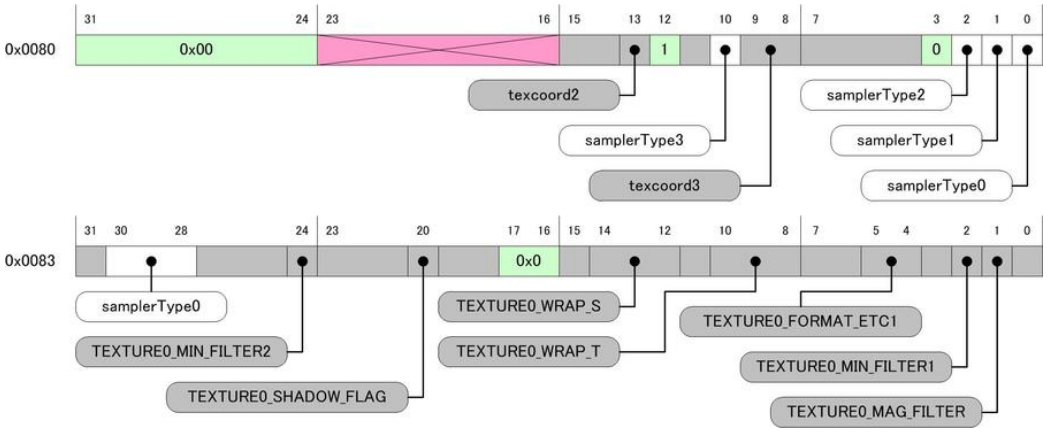
**Table 8-29. Names and Their Reserved Uniforms (Shadow Texture Setting Register)**

| Name | Bits | Description |
|---|---|---|
| perspectiveShadow | 1 | The value set in `dmp_Texture[0].perspectiveShadow`.<br>`0x0`: GL_TRUE<br>`0x1`: GL_FALSE<br>**Note:** Pay close attention to these values. |
| shadowZBias | 23 | The value set in `dmp_Texture[0].shadowZBias`, after it is then converted to a 23-bit unsigned fixed-point number.<br>This uses the upper 23 bits of the result of converting to a 24-bit fixed-point number. For more information about how these values are converted, see 8.9.14. Conversion to a 24-Bit Signed Fixed-Point Number With 24 Fractional Bits. |

## 8.8.6.2. Texture Sampler Type Setting Registers (0x0080, 0x0083)

The registers for setting the texture sampler types have the following layout. Note that the other bits in these registers are used for separate settings.

**Figure 8-41. Bit Layout of Texture Sampler Type Setting Registers (0x0080, 0x0083)**




The names in the bit layout correspond to the following reserved uniforms. The following table also shows the number of bits and possible values of each name.

**Table 8-30. Names and Their Reserved Uniforms (Texture Sampler Type Setting Registers)**

| Name | Bits | Description |
|---|---|---|
| samplerType0 | 1 + 3 | The values set to the corresponding bits in registers 0x0080 and 0x0083 change depending on the value set to dmp_Texture[0].samplerType.<br>When the uniform is GL_FALSE, register 0x0080 bit [0:0] and register 0x0083 bits [30:28] are all set to 0x0.<br>When the uniform has a setting other than GL_FALSE, register 0x0080 bit [0:0] is set to 0x1, and register 0x0083 bits [30:28] are set to the following values.<br>0x0 : GL_TEXTURE_2D<br>0x1 : GL_TEXTURE_CUBE_MAP<br>0x2 : GL_TEXTURE_SHADOW_2D_DMP<br>0x3 : GL_TEXTURE_PROJECTION_DMP<br>0x4 : GL_TEXTURE_SHADOW_CUBE_DMP |
| samplerType1 | 1 | The value set in dmp_Texture[1].samplerType.<br>0x0 : GL_FALSE<br>0x1 : GL_TEXTURE_2D |
| samplerType2 | 1 | The value set in dmp_Texture[2].samplerType.<br>0x0 : GL_FALSE<br>0x1 : GL_TEXTURE_2D |
| samplerType3 | 1 | The value set in dmp_Texture[3].samplerType.<br>0x0 : GL_FALSE<br>0x1 : GL_TEXTURE_PROCEDURAL_DMP |

The dmp_Texture[0].samplerType, dmp_Texture[1].samplerType, and dmp_Texture[2].samplerType settings are generated not via the NN_GX_STATE_FSUNIFORM state flag, but rather when the glDrawElements or glDrawArrays() function is called.

Write 1 to register 0x0080 bit [16:16] to clear all texture caches (both Level 1 and Level 2). When doing so, write 0 to all of the bits [23:17] of register 0x0080. Texture caches must be cleared when the texture unit settings have changed. To clear the L1 texture cache built into each individual texture unit, the texture units must be enabled beforehand. To enable a texture unit, set its sampler type to something other than GL_FALSE.

The texture unit enable command must be issued separately and in advance of the texture cache clear command. The texture cache clear operation uses the same register that holds the texture unit enable/disable settings. If the same single command performs both a bit write that enables a texture unit and a bit write that clears texture cache, the texture cache clear is not performed properly. On the other hand, the same is not true when disabling texture units. If the same single command performs both a bit write to disable a texture unit and a bit write to clear texture cache, the texture cache clear is performed properly.

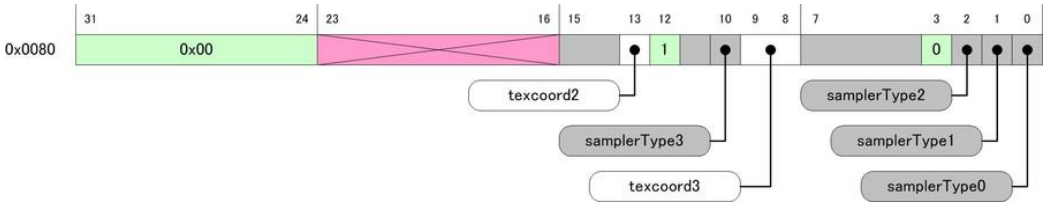
When you use a command to set other bits in register 0x0080 and you have no need to clear texture cache, set the byte enable to 0xB to ensure that you do not access bits [23:16] of the register.

There are two situations where texture cache must be cleared: when changes have been made to the texture address setting registers (0x0085, 0x0086, 0x0087, 0x0088, 0x0089, 0x008A, 0x0095, 0x009D), and when texture data has been reloaded. Even if no changes have been made to the texture addresses or data itself and only the format has changed, you must clear the caches.

## 8.8.6.3. Texture Coordinate Selection Setting Register (0x0080)

The register for setting the selection of what texture coordinates to input to the texture units has the following layout. Note that the other bits in this register are used for separate settings.

**Figure 8-42. Bit Layout of Texture Coordinate Selection Setting Register (0x0080)**




The names in the bit layout correspond to the following reserved uniforms. The following table also shows the number of bits and possible values of each name.

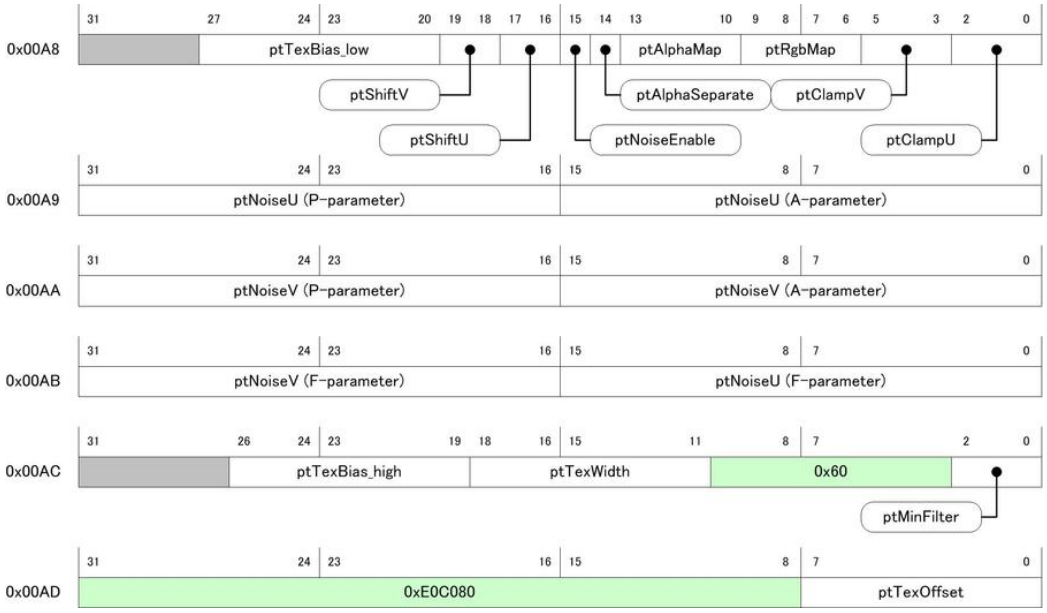
**Table 8-31. Names and Their Reserved Uniforms (Texture Coordinate Selection Setting Register)**

| Name | Bits | Description |
|---|---|---|
| texcoord2 | 1 | The value set in `dmp_Texture[2].texcoord`.<br>`0x0` : GL_TEXTURE2<br>`0x1` : GL_TEXTURE1 |
| texcoord3 | 2 | The value set in `dmp_Texture[3].texcoord`.<br>`0x0` : GL_TEXTURE0<br>`0x1` : GL_TEXTURE1<br>`0x2` : GL_TEXTURE2 |

## 8.8.6.4. Procedural Texture Setting Registers (0x00A8 – 0x00AD)

The registers for setting the reserved uniforms involved with procedural texture settings have the following bit layout.

**Figure 8-43. Bit Layout of Procedural Texture Setting Registers (0x00A8 – 0x00AD)**




The names in the bit layout correspond to the following reserved uniforms. The following table also shows the number of bits and possible values of each name.

**Table 8-32. Names and Their Reserved Uniforms (Procedural Texture Setting Registers)**

| Name | Bits | Description |
|---|---|---|
| ptRgbMap<br>ptAlphaMap | 4 | From the top, the values set in `dmp_Texture[3].ptRgbMap` and `dmp_Texture[3].ptAlphaMap`.<br>`0x0` : `GL_PROCTEX_U_DMP`<br>`0x1` : `GL_PROCTEX_U2_DMP`<br>`0x2` : `GL_PROCTEX_V_DMP`<br>`0x3` : `GL_PROCTEX_V2_DMP`<br>`0x4` : `GL_PROCTEX_ADD_DMP`<br>`0x5` : `GL_PROCTEX_ADD2_DMP`<br>`0x6` : `GL_PROCTEX_ADDSQRT2_DMP`<br>`0x7` : `GL_PROCTEX_MIN_DMP`<br>`0x8` : `GL_PROCTEX_MAX_DMP`<br>`0x9` : `GL_PROCTEX_RMAX_DMP` |
| ptAlphaSeparate | 1 | The value set in `dmp_Texture[3].ptAlphaSeparate`.<br>`0x0`: `GL_FALSE`<br>`0x1`: `GL_TRUE` |
| ptClampU<br>ptClampV | 3 | From the top, the values set in `dmp_Texture[3].ptClampU` and `dmp_Texture[3].ptClampV`.<br>`0x0` : `GL_CLAMP_TO_ZERO_DMP`<br>`0x1` : `GL_CLAMP_TO_EDGE`<br>`0x2` : `GL_SYMMETRICAL_REPEAT_DMP`<br>`0x3` : `GL_MIRRORED_REPEAT`<br>`0x4` : `GL_PULSE_DMP` |
| ptShiftU<br>ptShiftV | 2 | From the top, the values set in `dmp_Texture[3].ptShiftU` and `dmp_Texture[3].ptShiftV`.<br>`0x0` : `GL_NONE_DMP`<br>`0x1` : `GL_ODD_EDGE`<br>`0x2` : `GL_EVEN_DMP` |
| ptMinFilter | 3 | The value set in `dmp_Texture[3].ptMinFilter`.<br>`0x0` : `GL_NEAREST`<br>`0x1` : `GL_LINEAR`<br>`0x2` : `GL_NEAREST_MIPMAP_NEAREST`<br>`0x3` : `GL_LINEAR_MIPMAP_NEAREST`<br>`0x4` : `GL_NEAREST_MIPMAP_LINEAR`<br>`0x5` : `GL_LINEAR_MIPMAP_LINEAR` |
| ptTexOffset<br>ptTexWidth | 8 | From the top, the values set in `dmp_Texture[3].ptTexOffset` and `dmp_Texture[3].ptTexWidth`. These are identical to the uniforms' setting values. |
| ptTexBias_high<br>ptTexBias_low | 8 | This setting is the result of taking the value set in `dmp_Texture[3].ptTexBias`, converting it to a 16-bit floating-point number, and dividing it into two segments (upper and lower) of 8 bits each. For more information about how these values are converted, see 8.9.2. Converting to a 16-Bit Floating-Point Number. |
| ptNoiseEnable | 1 | The value set in `dmp_Texture[3].ptNoiseEnable`.<br>`0x0`: `GL_FALSE`<br>`0x1`: `GL_TRUE` |
| ptNoiseU (F-parameter)<br>ptNoiseU (P-parameter)<br>ptNoiseU (A-parameter) | 16 | From the top, the first through third elements of the values set in `dmp_TexEnv[3].ptNoiseV`.<br>For the first and second elements, the setting is the result of taking the uniform value and converting it to a 16-bit floating-point number. For more information about how these values are converted, see 8.9.2. Converting to a 16-Bit Floating-Point Number.<br>For the third element, the setting is the result of taking the uniform value and converting it to a signed 16-bit fixed-point number with 12 fractional bits (using two's complement to represent negative values). For more information about how these values are converted, see 8.9.10. Conversion to a 16-Bit Signed Fixed-Point Number With 12 Fractional Bits. |
| | | From the top, the first through third elements of the values set in |

| | | |
|---|---|---|
| ptNoiseV (F-parameter)<br>ptNoiseV (P-parameter)<br>ptNoiseV (A-parameter) | 16 | dmp_TexEnv[3].ptNoiseU.<br>For the first and second elements, the setting is the result of taking the uniform value and converting it to a 16-bit floating-point number. For more information about how these values are converted, see 8.9.2. Converting to a 16-Bit Floating-Point Number.<br>For the third element, the setting is the result of taking the uniform value and converting it to a signed 16-bit fixed-point number with 12 fractional bits (using two's complement to represent negative values). For more information about how these values are converted, see 8.9.10. Conversion to a 16-Bit Signed Fixed-Point Number With 12 Fractional Bits. |

## 8.8.6.5. Procedural Texture Lookup Table Setting Registers (0x00AF, 0x00B0 – 0x00B7)

Procedural texture lookup tables specified by the dmp_Texture[3].ptSampler{RgbMap, AlphaMap, NoiseMap, R, G, B, A} reserved uniforms have two possible sizes. Lookup tables for RgbMap, AlphaMap, and NoiseMap are set using 128 data items and 128 delta values, while lookup tables for R, G, B, and A are set using 256 data items and 256 delta values.

The same register is used to set the tables regardless of the number of data items, and its bit layout is as follows.

**Figure 8-44. Bit Layout of Procedural Texture Lookup Table Setting Registers (0x00AF and Others)**

In register 0x00AF, set *Proc_Table* to the lookup table you are targeting and *Proc_Index* to the index at which to start setting the table. Index 0 indicates the first data item. The values set in *Proc_Table* correspond to the following lookup tables. Even though four bits have been prepared for this 3-bit setting, you must set bit [11:11] equal to 0 to configure a lookup table properly.

**Table 8-33. Ref_Table Values and Lookup Tables**

| Proc_Table | Targeted Lookup Table |
|---|---|
| 0x0 | Noise modulation table (NoiseMap) |
| 0x2 | RGB mapping F function (RgbMap) |
| 0x3 | Alpha mapping F function (AlphaMap) |
| 0x4 | Color lookup table color values (R, G, B, A) |
| 0x5 | Color lookup table delta values (R, G, B, A) |

The same registers are used for setting values in all the lookup tables, but the bit layout of the registers differs between the color lookup tables and the other lookup tables.

After setting the targeted lookup table and index in register 0x00AF, write the data to any register in the range 0x00B0 – 0x00B7. The results are the same no matter which of these registers is written to, and the index is incremented by one for each data item that is written. This holds true

for all the lookup tables. Also note that register 0x0080 bit [10:10] (samplerType3) must be set to 0x1 (procedural textures); otherwise, writes to the registers 0x00B0 – 0x00B7 are ignored.

#### Noise Modulation Table, RGB Mapping F Function, and Alpha Mapping F Function

To set registers 0x01C8 through 0x01CF, write the values derived from combining the $i^{th}$ data item and $(i + 128)^{th}$ delta value in the lookup table loaded by the glTexImage1D() function. Set *Proc_Value* to the data item converted to an unsigned 12-bit fixed-point number with 12 fractional bits, and set *Proc_Difference* to the delta value converted to a signed 12-bit fixed-point number with 11 fractional bits (with negative numbers expressed using two's complement). For more information about how these values are converted, see 8.9.13. Conversion to a 12-Bit Unsigned Fixed-Point Number With 12 Fractional Bits and 8.9.7 Conversion to a 12-Bit Signed Fixed-Point Number With 11 Fractional Bits. Because there are 128 data items, the specifiable range for *Proc_Index* is 0 through 127.

#### Color Lookup Table

The values to write to the registers have a different format depending on whether you are setting color values or delta values.

For color values, you must write a packed value derived from the $i^{th}$ elements of each of the RGBA lookup tables loaded by the glTexImage1D() function. The value to write is obtained as follows: Take each component and convert it by mapping the range from 0.0 to 1.0 to the unsigned 8-bit integers 0 through 255. Set the R component in *Proc_R*, the G component in *Proc_G*, the B component in *Proc_B*, and the A component in *Proc_A*. (For more information about how these values are converted, see 8.9.16. Conversion From a Floating-Point Number (Between 0 and 1) to an 8-Bit Unsigned Integer.)

For delta values, you must write a packed value derived from the $(i+256)^{th}$ elements of each of the RGBA lookup tables loaded by the glTexImage1D() function. The value to write is obtained by doing the following: take each component and convert it to a signed 8-bit fixed-point number with 7 fractional bits (using two's complement to represent negative values). Set the R component in *Proc_R*, the G component in *Proc_G*, the B component in *Proc_B*, and the A component in *Proc_A*. (For more information about how these values are converted, see 8.9.5. Conversion to an 8-Bit Signed Fixed-Point Number With 7 Fractional Bits.) Because there are 256 data items, the specifiable range for *Proc_Index* is 0 through 255.

## 8.8.6.6. Texture Resolution Setting Registers (0x0082, 0x0092, 0x009A)

The registers for setting the respective resolutions of the textures loaded in texture units 0 through 2 have the following layout.

**Figure 8-45. Bit Layout of Texture Resolution Setting Registers (0x0082, 0x0092, 0x009A)**

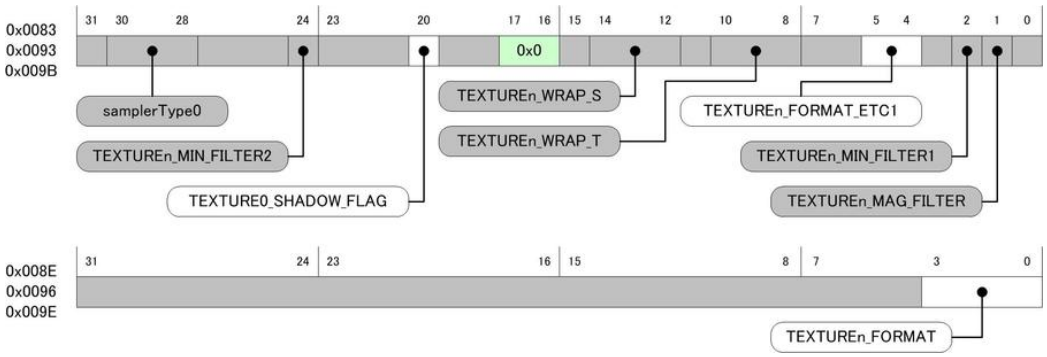
TEXTURE*n*_WIDTH and TEXTURE*n*_HEIGHT set the width and height of the texture loaded in texture unit *n* (where *n* is 0 through 2). The settings for texture unit 0 correspond to register

0x0082, texture unit 1 corresponds to register 0x0092, and texture unit 2 corresponds to register 0x009A.

## 8.8.6.7. Texture Format Setting Registers (0x0083, 0x008E, 0x0093, 0x0096, 0x009B, 0x009E)

The registers for setting the respective formats of the textures loaded in texture units 0 through 2 have the following layout.

**Figure 8-46. Bit Layout of Texture Format Setting Registers (0x0083, 0x008E and Others)**




The names in the bit layout correspond to the following texture format settings. The following table also shows the number of bits and possible values of each name. Settings for texture unit 0 correspond to registers 0x0083 and 0x008E. Settings for texture unit 1 correspond to registers 0x0093 and 0x0096. And settings for texture unit 2 correspond to registers 0x009B and 0x009E. Names that are not in the table correspond to bits used by other settings.

**Table 8-34. Names and Their Settings (Texture Format Setting Registers)**

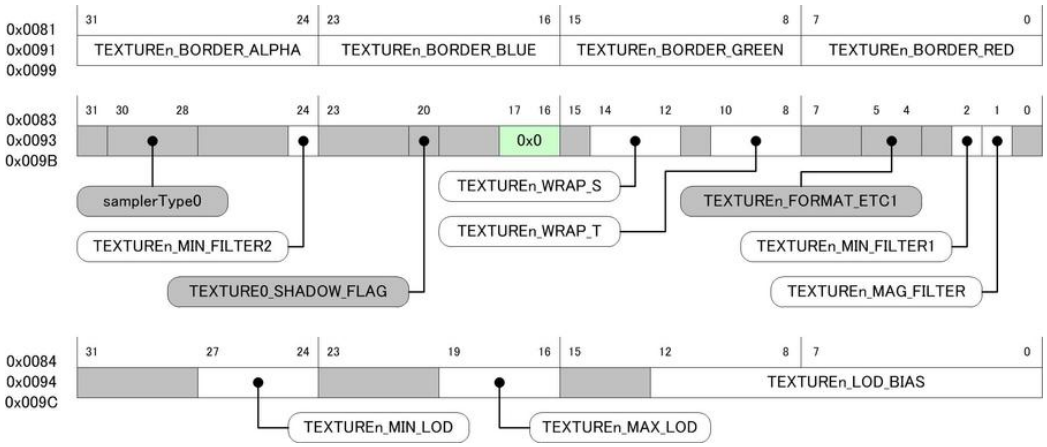
| Name | Bits | Description |
|---|---|---|
| TEXTUREn_FORMAT_ETC1 | 2 | This flag indicates whether GL_ETC1_RGB8_NATIVE_DMP is the texture format loaded in texture unit *n* (where *n* is 0 through 2).<br>0x00: Setting other than GL_ETC1_RGB8_NATIVE_DMP<br>0x02: GL_ETC1_RGB8_NATIVE_DMP |
| TEXTURE0_SHADOW_FLAG | 1 | This flag indicates whether the format of the texture loaded in texture unit 0 is GL_SHADOW_DMP (GL_SHADOW_NATIVE_DMP) or not (only register 0x0083). Register 0x0080, bits [2:0]<br>0x0: Setting other than GL_SHADOW_DMP (GL_SHADOW_NATIVE_DMP)<br>0x1: GL_SHADOW_DMP (GL_SHADOW_NATIVE_DMP) |
| TEXTUREn_FORMAT | 4 | The settings of the *format* and *type* parameters to the glTexImage2D() function, or the *internalformat* parameter to the glCompressedTexImage2D() function, for texture unit *n* (where *n* is 0 to 2).<br>0x0: GL_RGBA, GL_UNSIGNED_BYTE, GL_SHADOW_DMP, GL_UNSIGNED_INT, GL_GAS_DMP, GL_UNSIGNED_SHORT<br>0x1: GL_RGB, GL_UNSIGNED_BYTE<br>0x2: GL_RGBA, GL_UNSIGNED_SHORT_5_5_5_1<br>0x3: GL_RGB, GL_UNSIGNED_SHORT_5_6_5<br>0x4: GL_RGBA, GL_UNSIGNED_SHORT_4_4_4_4<br>0x5: GL_LUMINANCE_ALPHA, GL_UNSIGNED_BYTE<br>0x6: GL_HILO8_DMP, GL_UNSIGNED_BYTE<br>0x7: GL_LUMINANCE, GL_UNSIGNED_BYTE |

| | | |
|---|---|---|
| | | 0x8: GL_ALPHA, GL_UNSIGNED_BYTE<br>0x9: GL_LUMINANCE_ALPHA, GL_UNSIGNED_BYTE_4_4_DMP<br>0xA: GL_LUMINANCE, GL_UNSIGNED_4BITS_DMP<br>0xB: GL_ALPHA, GL_UNSIGNED_4BITS_DMP<br>0xC: GL_ETC1_RGB8_NATIVE_DMP<br>0xD: GL_ETC1_ALPHA_RGB8_A4_NATIVE_DMP<br>**Note:** The native formats use the same setting values as the corresponding non-native formats above.<br>**Note:** You cannot use GL_RGB, GL_UNSIGNED_BYTE (0x1) for a cube map. |

## 8.8.6.8. Texture Parameter Setting Registers (0x0081, 0x0083, 0x0084, and Others)

The registers for setting the texture parameters (such as wrapping mode and filters) for texture units 0 through 2 have the following layout.

**Figure 8-47. Bit Layout of Texture Parameter Setting Registers (0x0081, 0x0083, 0x0084 and Others)**




The names in the bit layout correspond to the following settings. The following table also shows the number of bits and possible values of each name. Settings for texture unit 0 correspond to registers 0x0081, 0x0083, and 0x0084; settings for texture unit 1 correspond to registers 0x0091, 0x0093, and 0x0094; and settings for texture unit 2 correspond to registers 0x0099, 0x009B, and 0x009C. Names that are not in the table correspond to bits used by other settings.

**Table 8-35. Names and Their Settings (Texture Parameter Setting Registers)**

| Name | Bits | Description |
|---|---|---|
| TEXTUREn_BORDER_RED<br>TEXTUREn_BORDER_GREEN<br>TEXTUREn_BORDER_BLUE<br>TEXTUREn_BORDER_ALPHA | 8 | These are the values of each component of the texture border color (GL_TEXTURE_BORDER_COLOR) for texture unit *n* (where *n* is 0 through 2).<br>To obtain the settings, take each component and convert it by mapping the range [0.0,1.0] to the unsigned 8-bit integers 0 through 255. For more information about how these values are converted, see 8.9.17. Conversion From a Floating-Point Number (Between 0 and 1) to an 8-Bit Unsigned Integer. |
| TEXTUREn_MAG_FILTER | 1 | Sets the texture magnification filter (GL_TEXTURE_MAG_FILTER) for texture unit *n* (where *n* is 0 to 2).<br>0x0: GL_NEAREST<br>0x1: GL_LINEAR |

| | | |
|---|---|---|
| TEXTUREn_MIN_FILTER1<br>TEXTUREn_MIN_FILTER2 | 1 | Sets the texture magnification filter (GL_TEXTURE_MAG_FILTER) for texture unit *n* (where *n* is from 0 through 2).<br>The following are the possible combinations of the two bits (FILTER1 and FILTER2). When different filter settings are present in a combination, the minimum and maximum LOD levels determine which setting is used.<br>(0x0, 0x0): GL_NEAREST, GL_NEAREST_MIPMAP_NEAREST<br>(0x0, 0x1): GL_NEAREST_MIPMAP_LINEAR<br>(0x1, 0x0): GL_LINEAR, GL_LINEAR_MIPMAP_NEAREST<br>(0x1, 0x1): GL_LINEAR_MIPMAP_LINEAR |
| TEXTUREn_WRAP_S<br>TEXTUREn_WRAP_T | 3 | Sets the wrapping modes (GL_TEXTURE_WRAP_S, GL_TEXTURE_WRAP_T) for texture unit *n* (where *n* is 0 to 2).<br>0x0: GL_CLAMP_TO_EDGE<br>0x1: GL_CLAMP_TO_BORDER<br>0x2 : GL_REPEAT<br>0x3: GL_MIRRORED_REPEAT |
| TEXTUREn_MIN_LOD | 4 | Sets the minimum LOD level (GL_TEXTURE_MIN_LOD) for texture unit *n* (where *n* is 0 through 2).<br>Set to 0 when LODs are not used, such as when the minification filter is GL_LINEAR or GL_NEAREST. When LODs are used, set the GL_TEXTURE_MIN_LOD value. If the value is 0 or less, set this to 0. |
| TEXTUREn_MAX_LOD | 4 | Sets the maximum LOD level for texture unit *n* (where *n* is 0 to 2).<br>Set to 0 when LODs are not used, such as when the minification filter is GL_LINEAR or GL_NEAREST. When LODs are used, set this to one less than the number of mipmap levels specified when the texture was loaded. |
| TEXTUREn_LOD_BIAS | 13 | Sets the LOD bias value (GL_TEXTURE_LOD_BIAS) for texture unit *n* (where *n* is 0 to 2).<br>The setting is a value converted into a signed 13-bit fixed-point number with 8 fractional bits (with negative numbers represented in two's complement). For more information about how these values are converted, see 8.9.8. Conversion to a 13-Bit Signed Fixed-Point Number With 8 Fractional Bits. |

### 8.8.6.9. Settings for Shadow Textures and Gas Textures

When using shadow textures (GL_SHADOW_DMP or GL_SHADOW_NATIVE_DMP) and 2D textures, set GL_TEXTURE_WRAP_S and GL_TEXTURE_WRAP_T to GL_CLAMP_TO_BORDER. When using shadow textures and cube map textures, set GL_TEXTURE_WRAP_S and GL_TEXTURE_WRAP_T to GL_CLAMP_TO_EDGE. When using either 2D textures or cube map textures, set GL_TEXTURE_MAG_FILTER and GL_TEXTURE_MIN_FILTER to GL_LINEAR and set register 0x0083 bit [20:20] (TEXTURE0_SHADOW_FLAG) to 1. Set this bit to 0 when using any format other than a shadow texture. Note that mipmaps cannot be applied to shadow textures.

When using gas textures (GL_GAS_DMP or GL_GAS_NATIVE_DMP), set GL_TEXTURE_WRAP_S and GL_TEXTURE_WRAP_T to GL_CLAMP_TO_EDGE and set in GL_TEXTURE_MAG_FILTER and GL_TEXTURE_MIN_FILTER to GL_NEAREST. Note that mipmaps cannot be applied to gas textures.

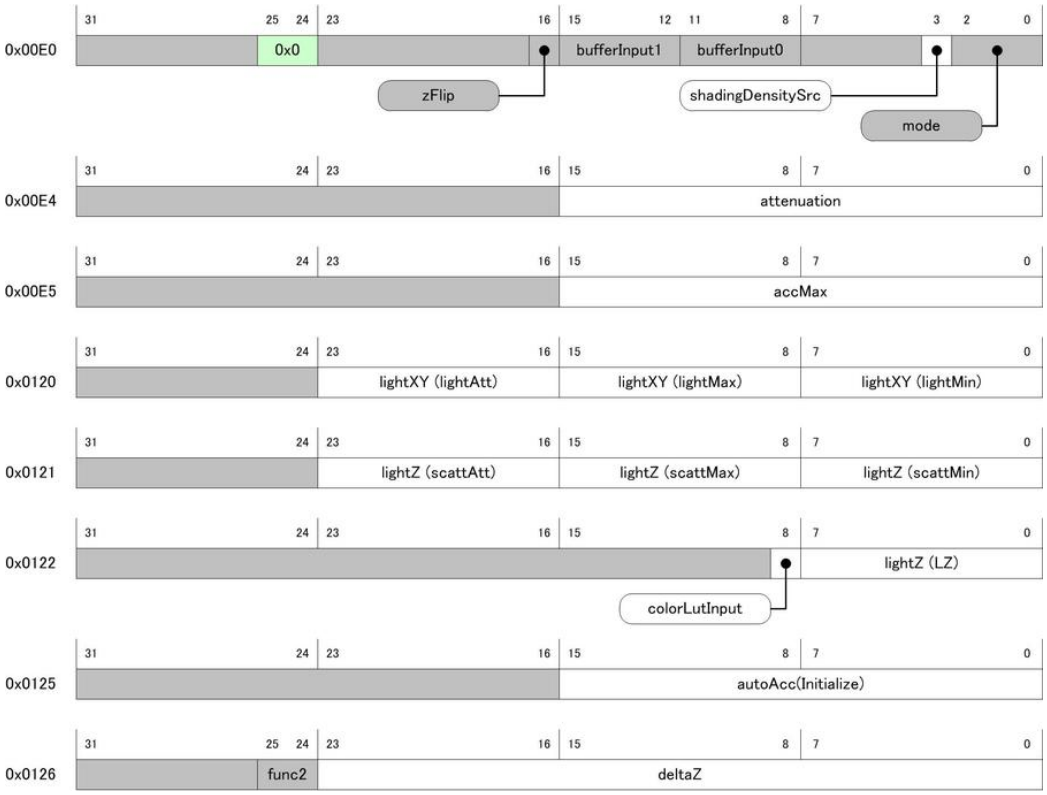
## 8.8.7. Gas Setting Registers (0x00E0, 0x00E4, 0x00E5, 0x0120 – 0x0126)

This section describes registers involved with reserved uniforms that make gas settings (uniforms whose names include `dmp_Gas`).

## 8.8.7.1. Gas Control Setting Registers (0x00E0, 0x00E4, 0x00E5, 0x0120 – 0x0122, 0x0125, 0x0126)

The registers for gas control settings have the following layout. Note that the other bits in register `0x00E0` are used for other settings, including fog settings.

**Figure 8-48. Bit Layout of Gas Control Setting Registers (0x00E0, 0x00E4, 0x00E5, 0x0120 – 0x0122, 0x0126)**




The names in the bit layout correspond to the following reserved uniforms. The following table also shows the number of bits and possible values of each name.

**Table 8-36. Names and Their Reserved Uniforms (Gas Control Setting Registers)**

| Name | Bits | Description |
|---|---|---|
| lightXY (lightMin)<br>lightXY (lightMax)<br>lightXY (lightAtt) | 8 | From the top, the first through third elements of the values set in `dmp_Gas.lightXY`.<br>To obtain the settings, take each individual uniform element value and convert it by mapping the range [`0.0`,`1.0`] to the unsigned, 8-bit integers `0` through 255. For more information about how these values are converted, see 8.9.16. Conversion From a Floating-Point Number (Between 0 and 1) to an 8-Bit Unsigned Integer. |
| lightZ (scattMin)<br>lightZ (scattMax)<br>lightZ (scattAtt) | 8 | From the top, the first through fourth elements of the values set in `dmp_Gas.lightZ`.<br>To obtain the settings, take each individual uniform element value and convert it by mapping the range [`0.0`,`1.0`] to the unsigned, 8-bit integers |

| | | |
|---|---|---|
| lightZ (LZ) | | 0 through 255. For more information about how these values are converted, see 8.9.16. Conversion From a Floating-Point Number (Between 0 and 1) to an 8-Bit Unsigned Integer. |
| deltaZ | 24 | The value set in dmp_Gas.deltaZ.<br>The setting is the result of converting the uniform value to an unsigned 24-bit fixed-point number with 8 fractional bits. For more information about how these values are converted, see 8.9.15. Conversion to an 8-Bit Signed Fixed-Point Number With 24 Fractional Bits. |
| accMax | 16 | The value set in dmp_Gas.accMax.<br>The setting is the result of converting the uniform value to a 16-bit floating-point number. For more information about how these values are converted, see 8.9.2. Converting to a 16-Bit Floating-Point Number. |
| autoAcc (Initialize) | 16 | The following procedures become necessary when dmp_Gas.autoAcc is set to GL_TRUE so that the inverse of the maximum density value is automatically calculated.<br>Zero-clear before the density-rendering pass, and call the nngxSetGasAutoAccumulationUpdate() function after the density-rendering pass finishes.<br>For more information, see the description below this table. |
| attenuation | 16 | The value set in dmp_Gas.attenuation.<br>The setting is the result of converting the uniform value to a 16-bit floating-point number. For more information about how these values are converted, see 8.9.2. Converting to a 16-Bit Floating-Point Number. |
| colorLutInput | 1 | The value set in dmp_Gas.colorLutInput.<br>0x0 : GL_GAS_DENSITY_DMP<br>0x1 : GL_GAS_LIGHT_FACTOR_DMP |
| shadingDensitySrc | 1 | The value set in dmp_Gas.shadingDensitySrc.<br>0x0: GL_GAS_DENSITY_DMP<br>0x1: GL_GAS_LIGHT_FACTOR_DMP |

### Automatically Calculating the Inverse of the Largest Density Value

If you set dmp_Gas.autoAcc to GL_TRUE, rendering is carried out in the gas density-rendering pass, and then the inverse of the maximum value of the D1 density values is automatically calculated and set in *accMax*.

Zero-clear the maximum D1 value before the density-rendering pass begins. Clear it by writing 0 to autoAcc (Initialize). (To clear using a value other than 0, make sure to convert that value to a 16-bit floating-point number before writing it. For more information about converting these values, see 8.9.2. Converting to a 16-Bit Floating-Point Number. After the density-rendering pass has finished, make sure to call the nngxSetGasAutoAccumulationUpdate() function, so that the calculated D1 maximum value is applied to accMax.

**Code 8-22. Definition of the nngxSetGasAutoAccumulationUpdate Function**

```
void nngxSetGasAutoAccumulationUpdate (GLint id);
```

This function, which uses the interrupt handler after completion of the $id^{th}$ accumulated command request in the bound command list object, applies the inverse of the maximum D1 value calculated during the gas density-rendering pass to *accMax* (the setting made by the dmp_Gas.accMax reserved uniform).

The command request specified by *id* must be a render command request. If you make sure to call this function on a command request that includes a density-rendering pass command, *accMax* will be updated correctly. Use the nngxSplitDrawCmdlist() function to separate density-rendering pass commands from shading pass commands. If both kinds of commands are included

in the same command request, *accMax* will not be updated before the shading pass. Also note that after you use this function to update *accMax*, you must not write any other value to *accMax* until the shading pass finishes.

Error GL_ERROR_806D_DMP occurs when a bound command list has the object name of 0. Error GL_ERROR_806E_DMP occurs when *id* is either 0 or less, larger than the number of accumulated command requests, or if the command request specified by *id* is not a render command request.

## 8.8.7.2. Shading Lookup Table Setting Registers (0x0123, 0x0124)

Shading lookup tables specified by the dmp_Gas.sampler{TR, TG, TB} reserved uniforms are set using eight data items and eight delta values.

The bit layout for the setting registers is as follows.

**Figure 8-49. Bit Layout of Shading Lookup Table Setting Registers (0x0123, 0x0124)**

In register 0x0123, set *Shading_Index* to the index at which to start setting the table. Index 0 indicates the first data item. Because there are 16 data items, the specifiable range for *Shading_Index* is 0 to 15.

After setting the index in register 0x0123, write the set of converted values to register 0x0124. The index is incremented by one for each data item that is written. However, you must note that the format of the values written to the register differs between the first 8 and latter 8 items in the shading lookup table, and that **the index set in the register differs from the index in the shading lookup table**. Also note that register 0x00E0 bits [2:0] (mode) must be set to 0x7 (indicating that the fog unit is set to gas mode). Otherwise, write operations to register 0x0124 are ignored.

For the first eight data items ($i < 8$), you must write a packed value derived from the $(i+8)^{th}$ elements of each of the lookup tables loaded by the glTexImage1D() function. The value to write is obtained by doing the following: Take each component and convert it to a signed 8-bit integer. Set the R component in Shading_R, the G component in Shading_G, and the B component in Shading_B. (For more information about how these values are converted, see 8.9.18. Conversion From a Floating-Point Number (Between -1 and 1) to an 8-Bit Signed Integer.)

For the latter eight data items ($i >= 8$), you must write a packed value derived from the $(i-8)^{th}$ elements of each of the lookup tables loaded by the glTexImage1D() function. The value to write is obtained by doing the following: Take each component, multiply it by 255, and then convert it to an unsigned 8-bit fixed-point number with 0 fractional bits. Set the R component in Shading_R, the G component in Shading_G, and the B component in Shading_B. (For more information about how these values are converted, see 8.9.11. Conversion to an 8-Bit Unsigned Fixed-Point Number With 0 Fractional Bits.)

Dummy commands must sometimes be inserted before commands that set the shading lookup table. In particular, immediately following any commands that set registers 0x0000 – 0x0035, registers 0x0100 – 0x013F, or registers at addresses not documented in this manual, you must insert 45 dummy commands before setting the gas shading lookup table. Any command that writes to a register outside these ranges can serve as a dummy command. Also note that after a command that sets the shading lookup table, you must insert one dummy command that sets

register 0x0100 and has a byte enable of 0.

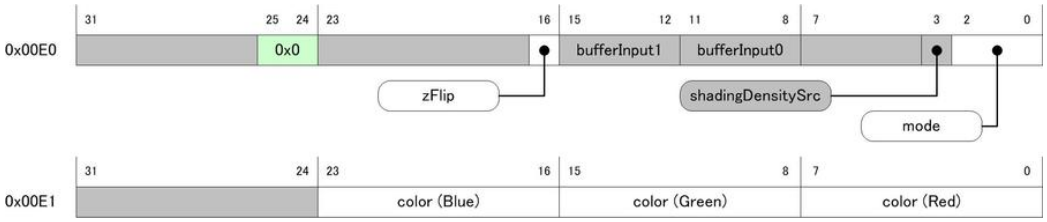
## 8.8.8. Fog Setting Registers (0x00E0, 0x00E1, 0x00E6, 0x00E8 - 0x00EF)

This section describes registers involved with reserved uniforms that make fog settings (uniforms whose names include dmp_Fog).

### 8.8.8.1. Fog Control Setting Registers (0x00E0, 0x00E1)

The registers for fog control settings have the following layout. Note that the other bits in register 0x00E0 are used for other settings (such as gas settings).

**Figure 8-50. Bit Layout of Fog Control Setting Registers (0x00E0, 0x00E1)**




The names in the bit layout correspond to the following reserved uniforms. The following table also shows the number of bits and possible values of each name.

**Table 8-37. Names and Their Reserved Uniforms (Fog Control Setting Registers)**

| Name | Bits | Description |
|---|---|---|
| mode | 3 | The value set in dmp_Fog.mode.<br>0x0 : GL_FALSE<br>0x5 : GL_FOG<br>0x7 : GL_GAS_DMP |
| zFlip | 1 | The value set in dmp_Fog.zFlip.<br>0x0: GL_FALSE<br>0x1: GL_TRUE |
| color (Red)<br>color (Green)<br>color (Blue) | 8 | From the top, the first through third elements of the values set in dmp_Fog.color. To obtain the settings, take each individual uniform element value and convert it by mapping the range [0.0,1.0] to the unsigned, 8-bit integers 0 through 255. For more information about how these values are converted, see 8.9.16. Conversion From a Floating-Point Number (Between 0 and 1) to an 8-Bit Unsigned Integer. |

### 8.8.8.2. Fog Lookup Table Setting Registers (0x00E6, 0x00E8 – 0x00EF)

The fog coefficient lookup table specified by the dmp_Fog.sampler reserved uniform is set using 128 data items and 128 delta values. The bit layout for the setting registers is as follows.

**Figure 8-51. Bit Layout of Fog Lookup Table Setting Registers (0x00E6, 0x00E8 – 0x00EF)**

In register `0x00E6`, set *Fog_Index* to the index at which to start setting the table. An index of `0` indicates the first data item, and `127` indicates the last data item.

To set registers `0x00E8` through `0x00EF`, write the values derived from combining the $i^{th}$ data item and $(i + 128)^{th}$ delta value in the lookup table loaded by the `glTexImage1D()` function. Set *Fog_Value* to the data item converted to an unsigned 11-bit fixed-point number with 11 fractional bits, and set *Fog_Difference* to the delta value converted to a signed 13-bit fixed-point number with 11 fractional bits (with negative numbers expressed using two's complement). For more information about how these values are converted, see 8.9.12. Conversion to an 11-Bit Unsigned Fixed-Point Number With 11 Fractional Bits and 8.9.9. Conversion to a 13-Bit Signed Fixed-Point Number With 11 Fractional Bits.

After setting the index in register `0x00E6`, write the sets of converted values to any register in the range `0x00E8` through `0x00EF`. The results are the same no matter which of these registers is written to, and the index is incremented by one for each data item that is written.

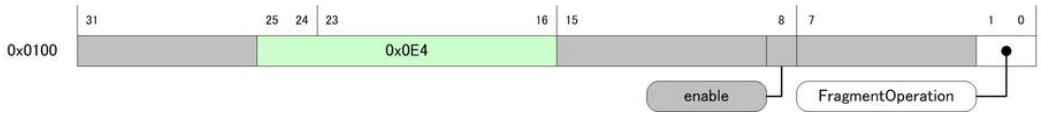
## 8.8.9. Per-Fragment Operations Setting Registers (0x0100 and Others)

This section describes registers involved with reserved uniforms that make settings for per-fragment operations (uniforms whose names include `dmp_FragOperation`).

### 8.8.9.1. Fragment Operations Mode Setting Register (0x0100)

The register for setting the fragment operations mode has the following layout. Note that the other bits in this register are used for logical operations and blending settings. When you change the fragment operations mode, you must also change the register settings described in 8.8.9.6. Framebuffer Access Control Setting Registers (0x0112 – 0x0115).

**Figure 8-52. Bit Layout of Fragment Operations Mode Setting Register (0x0100)**




The names in the bit layout correspond to the following reserved uniforms. The following table also shows the number of bits and possible values of each name.

**Table 8-38. Names and Their Reserved Uniforms (Fragment Operations Mode Setting Register)**

| Name | Bits | Description |
|---|---|---|
| | | The value set in `dmp_FragOperation.mode`. |

| | | |
|---|---|---|
| FragmentOperation | 2 | 0x0 : GL_FRAGOP_MODE_GL_DMP<br>0x1 : GL_FRAGOP_MODE_GAS_ACC_DMP<br>0x3 : GL_FRAGOP_MODE_SHADOW_DMP |

## 8.8.9.2. Shadow Attenuation Factor Setting Register (0x0130)

The register for setting the shadow attenuation factor has the following layout.

**Figure 8-53. Bit Layout of Shadow Factor Attenuation Setting Register (0x0130)**

**Table 8-39. Reserved Uniforms and Register for Setting the Shadow Attenuation Factor**

| Register | Bits | Description |
|---|---|---|
| 0x0130, bits [31:16] | 16 | Before the dmp_FragOperation.penumbraScale value is set to the register, its sign is flipped and it is converted to a 16-bit floating-point number.<br>For more information about how these values are converted, see 8.9.2. Converting to a 16-Bit Floating-Point Number. |
| 0x0130, bits [15:0] | 16 | The register setting is the sum of the dmp_FragOperation.penumbraScale and dmp_FragOperation.penumbraBias values, converted to a 16-bit floating-point number.<br>For more information about how these values are converted, see 8.9.2. Converting to a 16-Bit Floating-Point Number. |

## 8.8.9.3. w Buffer Setting Registers (0x004D, 0x004E, 0x006D)

The registers for w buffer settings have the following layout.

**Figure 8-54. Bit Layout of W-Buffer Setting Registers (0x004D, 0x004E, 0x006D)**

**Table 8-40. Reserved Uniforms and Registers for w Buffer Settings**

| Register | Bits | Description |
|---|---|---|
| 0x006D, | 1 | Set to 1 when the dmp_FragOperation.wScale setting value is 0, and to 0 when |

| | | |
|---|---|---|
| bit [0:0] | | the value is anything else. |
| 0x004D, bits [23:0] | 24 | Sets the scale value to apply to the clipping coordinate z value. The setting value is influenced by both the `dmp_FragOperation.wScale` setting value and the `glDepthRangef()` function's settings. Configure this setting as follows. |
| 0x004E bits [23:0] | 24 | Sets the bias value to apply to the clipping coordinate z value. The setting value is influenced by both the `dmp_FragOperation.wScale` setting value and the `glDepthRangef()` and `glPolygonOffset()` functions' settings. Configure this setting as follows. |

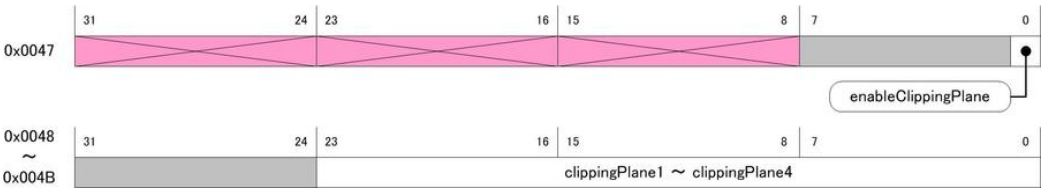
For register `0x004D` bits [`23:0`], if the `dmp_FragOperation.wScale` value is not `0`, the register setting is the result of taking the `dmp_FragOperation.wScale` value and flipping its sign. If the `dmp_FragOperation.wScale` value is `0`, obtain the register setting by taking the *zNear* and *zFar* parameters specified in the `glDepthRangef()` function and calculating (*zNear* – *zFar*). Convert the value to a 24-bit floating-point number before setting it in the register. For more information about how these values are converted, see 8.9.1. Converting to a 24-Bit Floating-Point Number.

For register `0x004E` bits [`23:0`], if the `dmp_FragOperation.wScale` value is not `0`, the register setting is `0`. If the `dmp_FragOperation.wScale` value is `0`, the register setting is the *zNear* parameter specified in the `glDepthRangef()` function. If `GL_POLYGON_OFFSET_FILL` has been enabled by the `glEnable()` function, the register setting is calculated by adding an offset derived from the *units* parameter specified in the `glPolygonOffset()` function. When the depth buffer format is 16 bits, the offset is the result of the calculation (*units* ÷ 65535). When the depth buffer format is 24 bits, the offset is the result of the calculation (*units* ÷ 16777215). Convert the value to a 24-bit floating-point number before setting it in the register. For more information about how these values are converted, see 8.9.1. Converting to a 24-Bit Floating-Point Number.

## 8.8.9.4. Clipping Setting Registers (0x0047 – 0x004B)

The registers for clipping settings have the following layout.

**Figure 8-55. Bit Layout of Clipping Setting Registers (0x0047 – 0x004B)**




The names in the bit layout correspond to the following reserved uniforms. The following table also shows the number of bits and possible values of each name.

**Table 8-41. Names and Their Reserved Uniforms (Clipping Setting Registers)**

| Name | Bits | Description |
|---|---|---|
| enableClippingPlane | 1 | The value set in `dmp_FragOperation.enableClippingPlane`.<br>`0x0`: `GL_FALSE`<br>`0x1`: `GL_TRUE` |
| clippingPlane1 through clippingPlane4 | 24 | The first through fourth elements of the values set in `dmp_FragOperation.clippingPlane`, converted into signed 24-bit floating-point numbers.<br>For more information about how these values are converted, see 8.9.1. |

| | | |
|---|---|---|
| | | Converting to a 24-Bit Floating-Point Number. |

## 8.8.9.5. Alpha Test Setting Register (0x0104)

The register for the alpha test settings has the following layout.

**Figure 8-56. Bit Layout of Alpha Test Setting Register (0x0104)**

The names in the bit layout correspond to the following reserved uniforms. The following table also shows the number of bits and possible values of each name.

**Table 8-42. Names and Their Reserved Uniforms (Alpha Test Setting Register)**

| Name | Bits | Description |
|---|---|---|
| enableAlphaTest | 1 | The value set in `dmp_FragOperation.enableAlphaTest`.<br>`0x0`: `GL_FALSE`<br>`0x1`: `GL_TRUE` |
| alphaTestFunc | 3 | The value set in `dmp_FragOperation.alphaTestFunc`.<br>`0x0` : `GL_NEVER`<br>`0x1` : `GL_ALWAYS`<br>`0x2` : `GL_EQUAL`<br>`0x3` : `GL_NOTEQUAL`<br>`0x4` : `GL_LESS`<br>`0x5` : `GL_LEQUAL`<br>`0x6` : `GL_GREATER`<br>`0x7` : `GL_GEQUAL` |
| alphaRefValue | 8 | To obtain the register setting, take the `dmp_FragOperation.alphaRefValue` value and convert it by mapping the range [`0.0`,`1.0`] to the unsigned 8-bit integers `0` through 255.<br>For more information about how these values are converted, see 8.9.16. Conversion From a Floating-Point Number (Between 0 and 1) to an 8-Bit Unsigned Integer. |

## 8.8.9.6. Framebuffer Access Control Setting Registers (0x0112 – 0x0115)

The registers for framebuffer access control settings have the following layout. These registers must sometimes be changed when certain functions are called, and when certain reserved uniform values are changed.

**Figure 8-57. Bit Layout of Framebuffer Access Control Setting Registers (0x0112 – 0x0115)**

**Table 8-43. Reserved Uniforms and Registers for Framebuffer Access Control Settings**

| Register | Bits | Description |
|---|---|---|
| colorRead 0x0110, bits [3:0] | 4 | Set `0x0F` when color buffer read operations are needed, and set to `0` otherwise.<br><br>Color buffer reads are needed when any of the following conditions are met.<br>• The reserved uniform `dmp_FragOperation.mode` is set to anything other than `GL_FRAGOP_MODE_GL_DMP`.<br>• The `glColorMask()` function currently permits writing of one or more components, and the `glEnable()` function has enabled `GL_BLEND`. Also, when blending and a blend factor referencing a DST color has been set using `glBlendFunc` or `glBlendFuncSeparate`.<br>• The `glColorMask()` function currently permits writing of one or more components, and the `glEnable()` function has enabled `GL_BLEND`. Also, when a logical operator referencing a DST color has been selected using `glLogicOp`.<br>• The `glColorMask()` function currently permits writing of one or more components, and also prohibits writing of one or more components. |
| colorWrite 0x0113, bits [3:0] | 4 | Set `0x0F` when you need to write to the color buffer, and set to `0` otherwise.<br><br>Color buffer writes are needed when any of the following conditions are met.<br>• The reserved uniform `dmp_FragOperation.mode` is set to anything other than `GL_FRAGOP_MODE_GL_DMP`.<br>• The `glColorMask()` function currently permits writing of one or more components. |
| depthRead 0x0114, bits [1:0] | 2 | Set bit [`1:1`] to `1` when depth buffer reads are needed, and set bit [`0:0`] to `1` when stencil buffer reads are needed. Set to `0` if not needed.<br><br>Depth buffer reads are needed when any of the following conditions are met.<br>• The reserved uniform `dmp_FragOperation.mode` is set to `GL_FRAGOP_MODE_GAS_ACC_DMP`.<br>• The reserved uniform `dmp_FragOperation.mode` is set to `GL_FRAGOP_MODE_GL_DMP`, the `glEnable()` function has enabled `GL_DEPTH_TEST`, and the `glDepthMask()` function is set to `GL_TRUE`.<br>• The reserved uniform `dmp_FragOperation.mode` is set to `GL_FRAGOP_MODE_GL_DMP`, the `glEnable()` function has enabled `GL_DEPTH_TEST`, and the `glColorMask()` function currently permits writing of one or more components.<br><br>Stencil buffer reads are needed when any of the following conditions are met.<br>• The reserved uniform `dmp_FragOperation.mode` is set to `GL_FRAGOP_MODE_GAS_ACC_DMP`. |

| | | |
|---|---|---|
| | | • The reserved uniform `dmp_FragOperation.mode` is set to `GL_FRAGOP_MODE_GL_DMP`, the `glEnable()` function has enabled `GL_STENCIL_TEST`, and the `glStencilMask()` function is set to a nonzero value.<br>• The reserved uniform `dmp_FragOperation.mode` is set to `GL_FRAGOP_MODE_GL_DMP`, the `glEnable()` function has enabled `GL_DEPTH_TEST`, and the `glColorMask()` function currently permits writing of one or more components. |
| depthWrite 0x0115, bits [1:0] | 2 | Set bit [`1:1`] to 1 when you need to write to the depth buffer , and set bit [`0:0`] to 1 when you need to write to the stencil buffer. Set to `0` if not needed.<br><br>Depth buffer writes are needed when the following conditions are met.<br>• The reserved uniform `dmp_FragOperation.mode` is set to `GL_FRAGOP_MODE_GL_DMP`, the `glEnable()` function has enabled `GL_DEPTH_TEST`, and the `glDepthMask()` function is set to `GL_TRUE`.<br><br>Stencil buffer writes are needed when the following conditions are met.<br>• The reserved uniform `dmp_FragOperation.mode` is set to `GL_FRAGOP_MODE_GL_DMP`, the `glEnable()` function has enabled `GL_STENCIL_TEST`, and the `glStencilMask()` function is set to a nonzero value. |

The hardware does not support all possible combinations of read and write operations for the color buffer, depth buffer, and stencil buffer. If an unsupported combination is set, operations are undefined. See the following table for the supported combinations.

**Table 8-44. Combinations Supported by Hardware**

| colorRead | colorWrite | depthRead | depthWrite | Hardware Support |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | × |
| Non-zero | 0 | 0 | 0 | × |
| 0 | Non-zero | 0 | 0 | ○ |
| Non-zero | Non-zero | 0 | 0 | ○ |
| 0 | 0 | Non-zero | 0 | × |
| Non-zero | 0 | Non-zero | 0 | × |
| 0 | Non-zero | Non-zero | 0 | ○ |
| Non-zero | Non-zero | Non-zero | 0 | ○ |
| 0 | 0 | 0 | Non-zero | × |
| Non-zero | 0 | 0 | Non-zero | × |
| 0 | Non-zero | 0 | Non-zero | × |
| Non-zero | Non-zero | 0 | Non-zero | × |
| 0 | 0 | Non-zero | Non-zero | ○ |
| Non-zero | 0 | Non-zero | Non-zero | × |
| 0 | Non-zero | Non-zero | Non-zero | ○ |
| Non-zero | Non-zero | Non-zero | Non-zero | ○ |

When these registers are set to `0`, memory access to the corresponding buffers is restricted and the result is a boost in performance. For this reason it is preferable that these registers are set to `0` if possible. However, you can only set combinations that are supported by the hardware. If write access is disabled, buffer writes will not take place, even if the fragment operations settings require buffer writes. Similarly, if read access is disabled but the fragment operations settings

require buffer reads, undefined values will be read from the buffer.

You can disable read access of the color buffer only if the following conditions are met.

- The reserved uniform `dmp_FragOperation.mode` is set to `GL_FRAGOP_MODE_GL_DMP`.
- Bits [`11:8`] of the color mask setting register (`0x0107`) are set to `0x0` or `0xF`.

In addition to all of the preceding conditions, at least one of the following conditions must be met.

- Blending is enabled (bit [`8:8`] of register `0x0100` is `1`) and the settings for *srcRGB* and *srcAlpha* (bits [`19:16`] and [`27:24`] of register `0x101`) do not require destination color (that is, are not `0x4`, `0x5`, `0x8`, `0x9`, or `0xE`). In addition, *dstRGB* and *dstAlpha* (bits [`23:20`] and [`31:38`] of register `0x0101`) are set to `0x0` (`GL_ZERO`).
- Logical operations are enabled (bit [`8:8`] of register `0x0100` is `0`) and the `opcode` setting for register `0x0102` does not require destination color (that is, `0x0`, `0x3`, `0x4`, or `0x5`).

You can disable write access of the color buffer only if all the following conditions are met.

- The reserved uniform `dmp_FragOperation.mode` is set to `GL_FRAGOP_MODE_GL_DMP`.
- Bits [`11:8`] of the color mask setting register (`0x0107`) are set to `0x0`.

You can disable read access of the depth buffer only if the following conditions are met.

- The reserved uniform `dmp_FragOperation.mode` is set to `GL_FRAGOP_MODE_GL_DMP`.
- Either the depth test is disabled, or it is enabled but the comparison method does not require the depth buffer value.

Either all of the preceding conditions must be met, or the following conditions must be met.

- The reserved uniform `dmp_FragOperation.mode` is set to `GL_FRAGOP_MODE_SHADOW_DMP`.

You can disable write access of the depth buffer only if at least one of the following conditions is met.

- The depth test is disabled (bit [`0:0`] of register `0x0107` is set to `0x0`).
- The depth buffer masking setting is `GL_FALSE` (bit [`12:12`] of register `0x0107` is set to `0x0`).
- The reserved uniform `dmp_FragOperation.mode` is set to something other than `GL_FRAGOP_MODE_GL_DMP`.

You can disable read access of the stencil buffer only if the following conditions are met.

- The reserved uniform `dmp_FragOperation.mode` is set to `GL_FRAGOP_MODE_SHADOW_DMP`.

Either this condition must be met, or, if the reserved uniform `dmp_FragOperation.mode` is set

to GL_FRAGOP_MODE_GL_DMP, at least one of the following conditions must be met.

- The stencil test is disabled (bit [0:0] of register 0x0105 is set to 0x0).
- The stencil test is enabled (bit [0:0] of register 0x0105 is set to 0x1), but the comparison method does not require the stencil buffer value.
- The stencil test masking value (bits [31:24] of register 0x0105) is set to 0x00.

You can disable write access of the stencil buffer only if at least one of the following conditions is met.

- The stencil test is disabled (bit [0:0] of register 0x0105 is set to 0x0).
- The stencil test mask setting (bits [15:8] of register 0x0105) is set to 0x00.
- The stencil test *fail*, *zfail* and *zpass* parameters are all set to GL_KEEP (bits [2:0], [6:4], and [10:8] of register 0x0106 are all set to 0x0).
- The reserved uniform dmp_FragOperation.mode is set to something other than GL_FRAGOP_MODE_GL_DMP.

## 8.8.10. Viewport Setting Registers (0x0041 – 0x0044, 0x0068)

The registers that correspond to the viewport settings configured by the glViewport() function have the following layout. When you change these registers, you must sometimes also change the registers described in 8.8.16. Scissor Test Setting Registers (0x0065 – 0x0067).

**Figure 8-58. Bit Layout of Viewport Setting Registers (0x0041 – 0x0044, 0x0068)**

The names in the bit layout correspond to the following viewport settings. The following table also shows the number of bits and possible values of each name.

**Table 8-45. Names and Their Settings (Viewport Setting Registers)**

| Name | Bits | Description |
|---|---|---|
| VIEWPORT_WIDTH | 24 | The result of dividing the viewport width by 2 and then converting the quotient into a 24-bit floating-point number.<br>For more information about how these values are converted, see |

| | | |
|---|---|---|
| | | 8.9.1. Converting to a 24-Bit Floating-Point Number. |
| VIEWPORT_WIDTH_INV | 32 | The result of dividing 2 by the viewport width, converting the quotient into a 31-bit floating-point number, and, finally, shifting the value left by 1 bit.<br>For more information about how these values are converted, see 8.9.3. Converting to a 31-Bit Floating-Point Number. |
| VIEWPORT_HEIGHT | 24 | The result of dividing the viewport height by 2 and then converting the quotient into a 24-bit floating-point number.<br>For more information about how these values are converted, see 8.9.1. Converting to a 24-Bit Floating-Point Number. |
| VIEWPORT_HEIGHT_INV | 32 | The result of dividing 2 by the viewport height, converting the quotient into a 31-bit floating-point number, and, finally, shifting the value left by 1 bit.<br>For more information about how these values are converted, see 8.9.3. Converting to a 31-Bit Floating-Point Number. |
| VIEWPORT_X | 10 | Sets the x-coordinate of the viewport's origin. |
| VIEWPORT_Y | 10 | Sets the y-coordinate of the viewport's origin. |

## 8.8.11. Depth Test Setting Registers (0x0107, 0x0126)

The registers that correspond to depth test settings have the following layout. When you change these registers, you must sometimes also change the registers described in 8.8.9.6. Framebuffer Access Control Setting Registers (0x0112 – 0x0115).

**Figure 8-59. Bit Layout of Depth Test Setting Registers (0x0107, 0x0126)**

The names in the bit layout correspond to the following depth test settings. The following table also shows the number of bits and possible values of each name. Names that are not in the table correspond to bits used by other settings.

**Table 8-46. Names and Their Settings (Depth Test Setting Registers)**

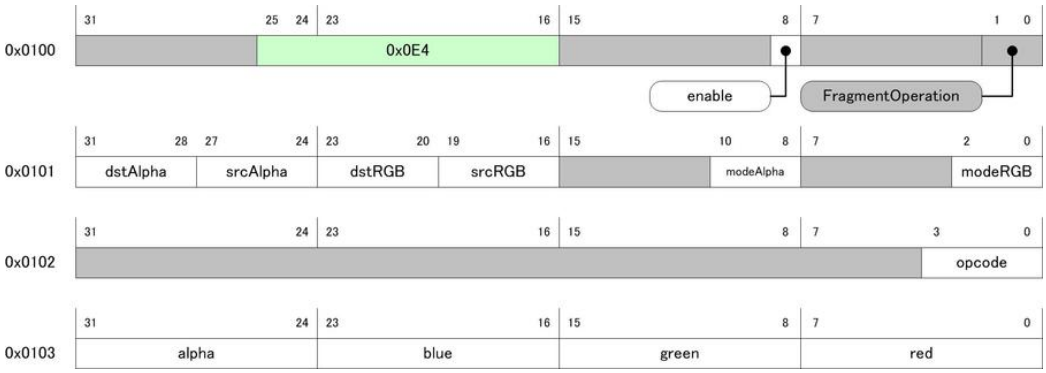
| Name | Bits | Description |
|---|---|---|
| enable | 1 | Sets whether to enable or disable depth tests. This setting is changed by passing `GL_DEPTH_TEST` to the `glEnable` or `glDisable()` function.<br>`0x0`: Depth tests disabled<br>`0x1`: Depth tests enabled |
| func | 3 | Sets the depth test comparison method. This setting is changed by the `glDepthFunc()` function.<br>`0x0`: `GL_NEVER`<br>`0x1`: `GL_ALWAYS`<br>`0x2`: `GL_EQUAL`<br>`0x3`: `GL_NOTEQUAL`<br>`0x4`: `GL_LESS`<br>`0x5`: `GL_LEQUAL`<br>`0x6`: `GL_GREATER` |

| | | |
|---|---|---|
| | | 0x7: GL_GEQUAL |
| flag | 1 | Depth buffer masking setting. This setting is changed by the glDepthMask() function.<br>0x0: GL_FALSE<br>0x1: GL_TRUE |
| func2 | 2 | Sets the depth test comparison method. This setting is changed by the glDepthFunc() function. However, this setting does not affect the normal depth test. It affects how the density information, D2, is calculated during the gas density-rendering pass.<br>0x0: GL_NEVER<br>0x1: GL_ALWAYS<br>0x2: GL_GREATER or GL_EQUAL<br>0x3: Any other value |

## 8.8.12. Logical Operations and Blending Setting Registers (0x0100 – 0x0103)

The following registers correspond to logical operations and blending settings. When you change these registers, you must sometimes also change the registers described in 8.8.9.6. Framebuffer Access Control Setting Registers (0x0112 – 0x0115).

**Figure 8-60. Bit Layout of Logical Operations and Blending Setting Registers (0x0100 – 0x0103)**




The names in the bit layout correspond to the following early depth test settings. The following table also shows the number of bits and possible values of each name.

**Table 8-47. Names and Their Settings (Logical Operations and Blending Setting Registers)**

| Name | Bits | Description |
|---|---|---|
| enable | 1 | Sets whether logical operations or blending is enabled (they are mutually exclusive). This setting is changed by passing either GL_BLEND or GL_LOGIC_OP to the glEnable or glDisable() function. If both are set as enabled, logical operations take priority. If both are set as disabled, blending takes priority.<br>0x1: Blending is enabled |
| srcRGB<br>dstRGB | | Sets the source and destination weighting coefficients. These settings are changed by the glBlendFunc() and glBlendFuncSeparate() functions.<br>When blending is disabled, set *srcRGB* and *srcAlpha* to 0x1, and set *dstRGB* and *dstAlpha* to 0x0.<br>When blending is enabled, you can set the following values.<br>0x0: GL_ZERO<br>0x1: GL_ONE<br>0x2: GL_SRC_COLOR<br>0x3: GL_ONE_MINUS_SRC_COLOR<br>0x4: GL_DST_COLOR |

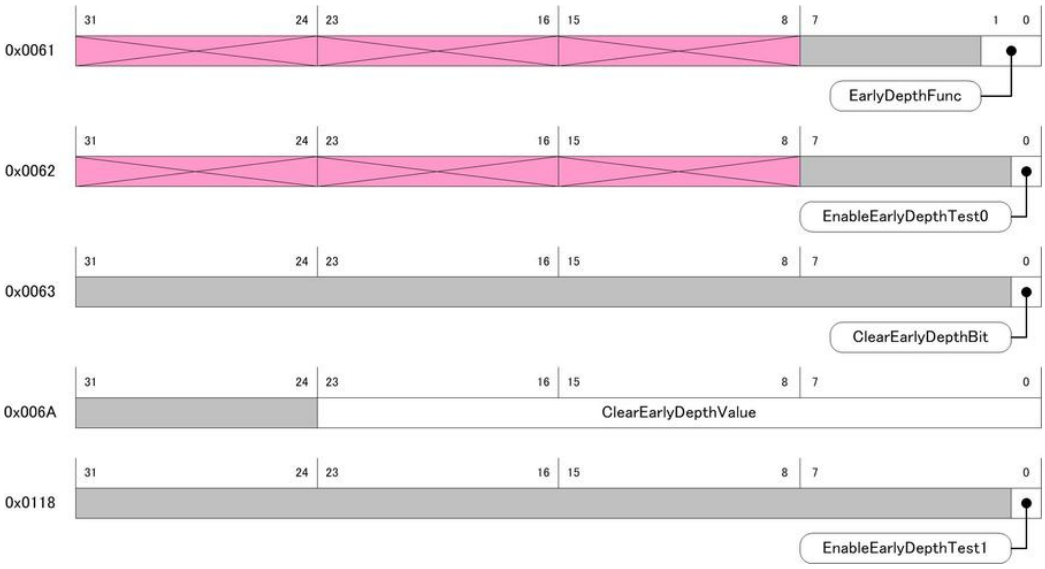
| | | |
|---|---|---|
| srcAlpha<br>dstAlpha | 4 | 0x5: GL_ONE_MINUS_DST_COLOR<br>0x6: GL_SRC_ALPHA<br>0x7: GL_ONE_MINUS_SRC_ALPHA<br>0x8: GL_DST_ALPHA<br>0x9: GL_ONE_MINUS_DST_ALPHA<br>0xA: GL_CONSTANT_COLOR<br>0xB: GL_ONE_MINUS_CONSTANT_COLOR<br>0xC: GL_CONSTANT_ALPHA<br>0xD: GL_ONE_MINUS_CONSTANT_ALPHA<br>0xE: GL_SRC_ALPHA_SATURATE |
| modeRGB<br>modeAlpha | 3 | Sets the blending equation. This setting is changed by the glBlendEquation() and glBlendEquationSeparate() functions.<br>When blending is disabled, set to 0x0.<br>When blending is enabled, you can set the following values.<br>0x0: GL_FUNC_ADD<br>0x1: GL_FUNC_SUBTRACT<br>0x2: GL_FUNC_REVERSE_SUBTRACT<br>0x3: GL_MIN<br>0x4: GL_MAX |
| red<br>green<br>blue<br>alpha | 8 | From the top, the red, green, blue, and alpha components of the constant color set by the glBlendColor() function.<br>To obtain the settings, take each individual component value and convert it by mapping the range [0.0,1.0] to the unsigned 8-bit integers 0 through 255. For more information about how these values are converted, see 8.9.16. Conversion From a Floating-Point Number (Between 0 and 1) to an 8-Bit Unsigned Integer. |
| opcode | 4 | Sets the logical operation. This setting is changed by the glLogicOp() function.<br>0x0: GL_CLEAR<br>0x1: GL_AND<br>0x2: GL_AND_REVERSE<br>0x3: GL_COPY<br>0x4: GL_SET<br>0x5: GL_COPY_INVERTED<br>0x6: GL_NOOP<br>0x7: GL_INVERT<br>0x8: GL_NAND<br>0x9: GL_OR<br>0xA: GL_NOR<br>0xB: GL_XOR<br>0xC: GL_EQUIV<br>0xD: GL_AND_INVERTED<br>0xE: GL_OR_REVERSE<br>0xF: GL_OR_INVERTED |

When logical operations are enabled, the settings in register 0x0101 (src*XXX*, dst*XXX*, and mode*XXX*) are ignored.

## 8.8.13. Early Depth Test Setting Registers (0x0061 – 0x0063, 0x006A, 0x0118)

The following registers correspond to early depth test settings. When you change these registers, you must sometimes also change the registers described in 8.8.9.6. Framebuffer Access Control Setting Registers (0x0112 – 0x0115) and 8.8.11. Depth Test Setting Registers (0x0107, 0x0126).

**Figure 8-61. Bit Layout of Early Depth Test Setting Registers (0x0061 – 0x0063, 0x006A, 0x0118)**




The names in the bit layout correspond to the following logical operations and blending settings. The following table also shows the number of bits and possible values of each name.

**Table 8-48. Names and Their Settings (Depth Test Setting Registers)**

| Setter Function | Bits | Description |
|---|---|---|
| EnableEarlyDepthTest0<br>EnableEarlyDepthTest1 | 1<br>1 | Sets whether to enable or disable early depth tests. This setting is changed by passing `GL_EARLY_DEPTH_TEST_DMP` to the `glEnable` or `glDisable()` function.<br>`0x0`: Early depth tests disabled |
| EarlyDepthFunc | 2 | Sets the comparison method. This setting is specified by the *`func`* parameter of the `glEarlyDepthFuncDMP()` function.<br>`0x0` : `GL_CLAMP_TO_EDGE`<br>`0x1` : `GL_CLAMP_TO_BORDER`<br>`0x2` : `GL_REPEAT`<br>`0x3` : `GL_MIRRORED_REPEAT` |
| ClearEarlyDepthBit | 1 | This is set to a value of `0x1` when the early depth buffer is cleared by passing an argument containing `GL_EARLY_DEPTH_BUFFER_BIT_DMP` to the `glClear()` function. |
| ClearEarlyDepthValue | 24 | Sets the clear value for the early depth buffer. This setting is specified by the *`depth`* parameter of the `glClearEarlyDepthDMP()` function. Set the register to a value identical to the value passed as an argument. |

## 8.8.14. Stencil Test Setting Registers (0x0105, 0x0106)

The following registers correspond to stencil test settings. When you change these registers, you must sometimes also change the registers described in 8.8.9.6. Framebuffer Access Control Setting Registers (0x0112 – 0x0115).

**Figure 8-62. Bit Layout of Stencil Test Setting Registers (0x0105, 0x0106)**

The names in the bit layout correspond to the following stencil test settings. The following table also shows the number of bits and possible values of each name.

**Table 8-49. Names and Their Settings (Stencil Test Setting Registers)**

| Name | Bits | Description |
|---|---|---|
| enable | 1 | Sets whether to enable or disable stencil tests. This setting is changed by passing GL_STENCIL_TEST to the glEnable or glDisable() function.<br>0x0: Stencil tests disabled |
| fb_mask | 8 | Sets the stencil buffer's masking value. This setting is specified by the *mask* parameter of the glStencilMask() function. Set the register to the lower 8 bits of the value passed as an argument. |
| func | 3 | Sets the comparison method. This setting is specified by the *func* parameter of the glStencilFunc() function.<br>0x0: GL_NEVER<br>0x1: GL_ALWAYS<br>0x2: GL_EQUAL<br>0x3: GL_NOTEQUAL<br>0x4: GL_LESS<br>0x5: GL_LEQUAL<br>0x6: GL_GREATER<br>0x7: GL_GEQUAL |
| ref | 8 | Sets the stencil test reference value. This setting is specified by the *ref* parameter of the glStencilFunc() function. Set the register to a value identical to the value passed as an argument. |
| mask | 8 | Sets the stencil buffer's masking value. This setting is specified by the *mask* parameter of the glStencilMask() function. Set the register to a value identical to the value passed as an argument. |
| fail | 3 | Sets how the stencil buffer content is changed when a fragment is eliminated by the stencil test. This setting is changed by the *fail* parameter of the glStencilOp() function.<br>0x0: GL_KEEP<br>0x1: GL_ZERO<br>0x2: GL_REPLACE<br>0x3: GL_INCR<br>0x4: GL_DECR<br>0x5: GL_INVERT<br>0x6: GL_INCR_WRAP<br>0x7: GL_DECR_WRAP |
| zfail | 3 | Sets how the stencil buffer content is changed when a fragment is eliminated by the depth test. This setting is changed by the *zfail* parameter of the glStencilOp() function. This setting and *fail* have the same value. |
| zpass | 3 | Sets how the stencil buffer content is changed when a fragment passes the depth test. This setting is changed by the *zpass* parameter of the glStencilOp() function. This setting and *fail* have the same value. |

## 8.8.15. Culling Setting Register (0x0040)

The following register corresponds to culling settings.

**Figure 8-63. Bit Layout of Culling Setting Register (0x0040)**

The names in the bit layout correspond to the following stencil test settings. The following table also shows the number of bits and possible values of each name.

**Table 8-50. Names and Their Settings (Culling Setting Register)**

| Name | Bits | Description |
|---|---|---|
| Culling | 2 | Set to `0x0` when culling has been disabled by calling `glDisable(GL_CULL_FACE)`.<br>The value set when culling has been enabled by calling `glEnable(GL_CULL_FACE)` is determined by the following combinations of arguments passed to the `glCullFace()` and `glFrontFace()` functions.<br>Set to `0x2` if `glCullFace` is `GL_FRONT` and `glFrontFace` is `GL_CW`, or if `glCullFace` is `GL_BACK` and `glFrontFace` is `GL_CCW`.<br>Set to `0x1` for any other case. |

## 8.8.16. Scissor Test Setting Registers (0x0065 – 0x0067)

The registers corresponding to scissor test settings have the following layout.

**Figure 8-64. Bit Layout of Scissor Test Setting Registers (0x0065 – 0x0067)**

The names in the bit layout correspond to the following scissor test settings. The following table also shows the number of bits and possible values of each name.

**Table 8-51. Names and Their Settings (Scissor Test Setting Registers)**

| Name | Bits | Description |
|---|---|---|
| enable | 2 | Sets whether to enable or disable scissor tests. This setting is changed by passing `GL_SCISSOR_TEST` to the `glEnable` or `glDisable()` function.<br>`0x0`: Scissor tests disabled<br>`0x3`: Scissor tests enabled |
| x | 10 | Sets the x-coordinate of the origin of the scissor box. This setting is specified by the *x* parameter of the `glScissor()` function.<br>Set to `0` when the scissor test is disabled.<br>Set identical to the *x* parameter when the scissor test is enabled, except when *x* is equal to or larger than the current color buffer width (in which case one less than the current color buffer width is set) or *x* is negative (in which case `0` is set). |
| y | 10 | Sets the y-coordinate of the origin of the scissor box. This setting is specified by the *y* parameter of the `glScissor()` function.<br>Set to `0` when the scissor test is disabled.<br>Set identical to the *y* parameter when the scissor test is enabled, except when *y* is |

| | | |
|---|---|---|
| | | equal to or larger than the current color buffer height (in which case one less than the current color buffer height is set) or *y* is negative (in which case `0` is set). |
| width | 10 | Sets the scissor box width. This setting is specified by the *width* parameter of the `glScissor()` function.<br>Set to one less than the current color buffer width when the scissor test is disabled.<br>Set to the result of (*x* + *width* –1) when the scissor test is enabled, except when the result is equal to or larger than the current color buffer width (in which case one less than the current color buffer width is set) or the result is negative (in which case `0` is set). |
| height | 10 | Sets the scissor box height. This setting is specified by the *height* parameter of the `glScissor()` function.<br>Set to one less than the current color buffer height when the scissor test is disabled.<br>Set to the result of (*y* + *height* –1) when the scissor test is enabled, except when the result is equal to or larger than the current color buffer height (in which case one less than the current color buffer height is set) or the result is negative (in which case `0` is set). |

## 8.8.17. Color Mask Setting Register (0x0107)

The following register corresponds to color mask settings made by the `glColorMask()` function. When you change these registers, you must sometimes also change the registers described in 8.8.9.6. Framebuffer Access Control Setting Registers (0x0112 – 0x0115).

**Figure 8-65. Bit Layout of Color Mask Setting Register (0x0107)**

The names in the bit layout correspond to the following color mask settings. The following table also shows the number of bits and possible values of each name. Names that are not in the table correspond to bits used by the registers described in 8.8.11. Depth Test Setting Registers (0x0107, 0x0126).

**Table 8-52. Names and Their Settings (Color Mask Setting Register)**

| Name | Bits | Description |
|---|---|---|
| red | 1 | Sets the color mask's red component. This setting is specified by the *red* parameter of the `glColorMask()` function.<br>`0x0`: GL_FALSE<br>`0x1`: GL_TRUE |
| green | 1 | Sets the color mask's green component. This setting is specified by the *green* parameter of the `glColorMask()` function.<br>`0x0`: GL_FALSE<br>`0x1`: GL_TRUE |
| blue | 1 | Sets the color mask's blue component. This setting is specified by the *blue* parameter of the `glColorMask()` function.<br>`0x0`: GL_FALSE<br>`0x1`: GL_TRUE |
| alpha | 1 | Sets the color mask's alpha component. This setting is specified by the *alpha* parameter of the `glColorMask()` function.<br>`0x0`: GL_FALSE<br>`0x1`: GL_TRUE |

## 8.8.18. Block Format Setting Register (0x011B)

The following register corresponds to the block format setting.

**Table 8-53. Block Format Settings and Register**

| Setter Function | Register | Description |
|---|---|---|
| `glRenderBlockModeDMP` | 0x011B, bits [0:0] | `0x0`: GL_RENDER_BLOCK8_MODE_DMP<br>`0x1`: GL_RENDER_BLOCK32_MODE_DMP |

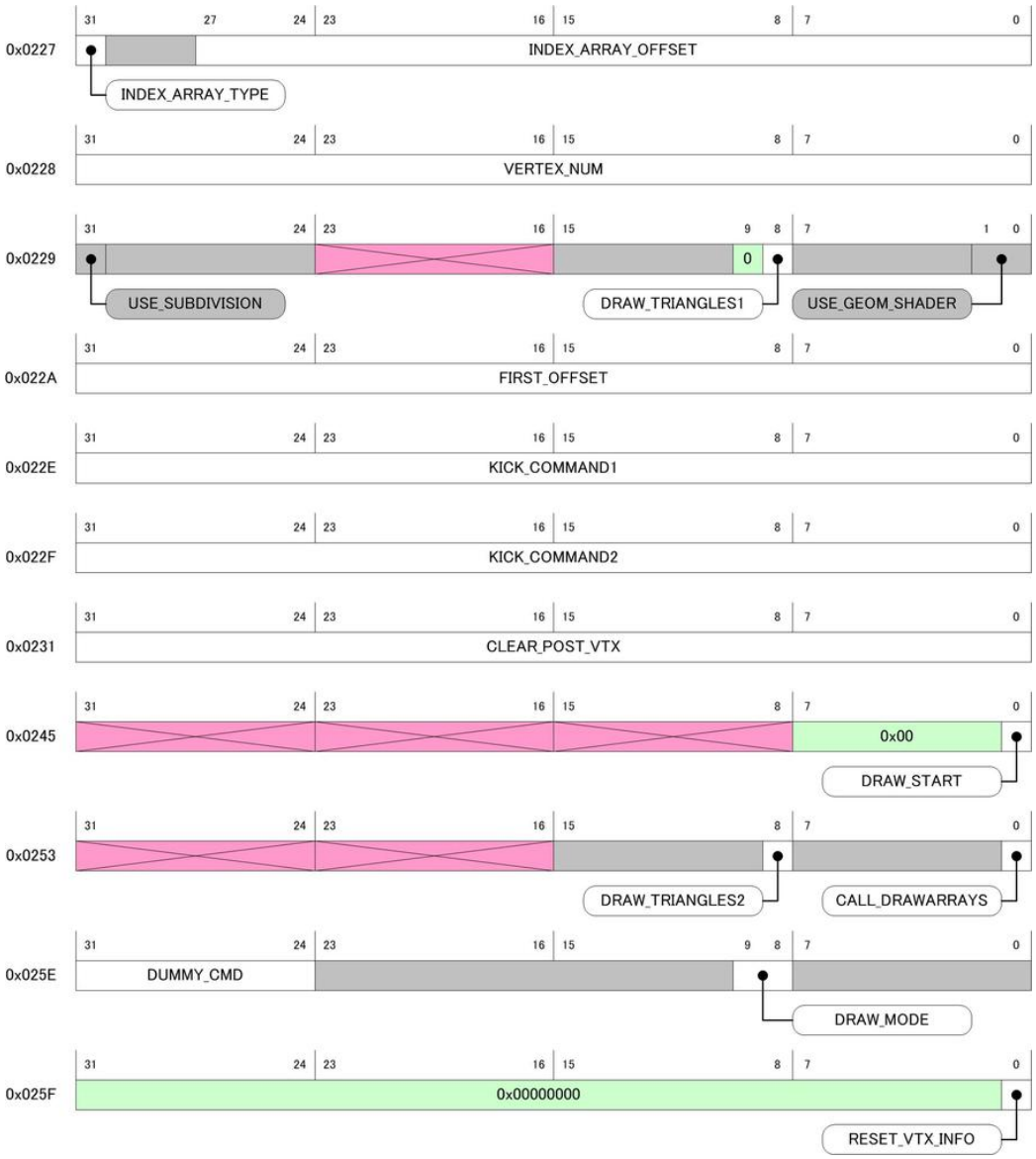
## 8.8.19. Registers Involved With the Rendering API (0x0227 – 0x022A and Others)

The `glDrawElements()` and `glDrawArrays()` functions validate states when they are called. This generates commands that set the various registers relating to the various states. But in addition to generating commands through validation, these functions set registers required for rendering itself.

### 8.8.19.1. Settings When Vertex Buffers Are Used

The following registers are set when vertex buffers are used with rendering. All these commands must be set before the render start command, unless indicated otherwise.

**Figure 8-66. Bit Layout of Setting Registers When Vertex Buffers Are Used for Rendering**




The names in the bit layout correspond to the following settings.

**Table 8-54. Names and Their Settings (Setting Registers When Vertex Buffers Are Used for Rendering)**

| Name | Bits | Description |
|---|---|---|
| DRAW_MODE<br>DRAW_TRIANGLES1<br>DRAW_TRIANGLES2 | 2<br>1<br>1 | Sets the rendering mode. This setting is specified by the *mode* parameter of the `glDrawElements` or `glDrawArrays()` function. This does not need to be reset unless settings change.<br>0x0: `GL_TRIANGLES` (for `glDrawArrays()` function only)<br>0x1: `GL_TRIANGLE_STRIP`<br>0x2: `GL_TRIANGLE_FAN`<br>0x3: `GL_GEOMETRY_PRIMITIVE_DMP` (However, with the `glDrawElements()` function, `GL_TRIANGLES` is specified if `DRAW_TRIANGLES1` and `DRAW_TRIANGLES2` have both been set to 1.) |
| CALL_DRAWARRAYS | 1 | Always set to `0` with `glDrawElements` and always set to 1 with `glDrawArrays`. Because the initial setting is `0` when the library is initialized, set this to 1 when the `glDrawArrays()` function is called and reset to `0` when the function ends.<br>While this is set to 1, there is a possibility that registers |

| | | |
|---|---|---|
| | | other than 0x0200 through 0x0254 and 0x0280 through 0x02DF might not be set properly. |
| INDEX_ARRAY_OFFSET | 28 | Sets the address offset to the vertex index array. This is the offset from the base address shared by all vertex arrays. The base address is set in bits [28:1] of register 0x0200. If you multiply the base address by 16 and add this offset, the result is identical to adding the *indices* parameter of the glDrawElements() function to the starting address of the vertex buffer allocated by the glBufferData() function.<br>When using the glDrawArrays() function, set this to 0x20 if the following conditions are met, and set to 0 otherwise.<br>• If VERTEX_NUM is greater than 0x10: ((VERTEX_NUM − 0x10)×2 + (ARRAY_BASE_ADDR << 4 )) & 0xFFF >= 0xFE0<br>• If VERTEX_NUM is less than or equal to 0x10: (ARRAY_BASE_ADDR << 4 ) & 0xFFF >= 0xFE0<br>The setting value for VERTEX_NUM is bits [31:0] of register 0x0228, and for ARRAY_BASE_ADDR, bits [28:1] of register 0x0200.<br>This does not need to be reset unless settings change. |
| INDEX_ARRAY_TYPE | 1 | Sets the vertex index type. Set to 1 when the *type* parameter of the glDrawElements() function is GL_UNSIGNED_SHORT, and set to 0 when the same argument is GL_UNSIGNED_BYTE.<br>Always set to 1 when using the glDrawArrays() function. This does not need to be reset unless settings change. |
| VERTEX_NUM | 32 | Sets the number of vertices to render. This does not need to be reset unless settings change. Operation is undefined when set to 0, so do not set this to 0. |
| FIRST_OFFSET | 32 | When rendering with the glDrawArrays() function, this is set to the value of the *first* parameter. This does not need to be reset unless settings change. |
| RESET_VTX_INFO | 1 | When a value of 1 is written to this bit, the vertex information for indices 0, 1, and 2 (which form a triangle) is reset.<br>When passing GL_GEOMETRY_PRIMITIVE_DMP as the rendering mode argument, or when rendering with the glDrawElements() function and passing GL_TRIANGLES, this information does not need to be set.<br>When rendering with the glDrawArrays() function and passing GL_TRIANGLES, this information does not need to be reset when making the second or subsequent calls, as long as the number of vertices is a multiple of three and the function is called successively in the same mode (glDrawElements is not called in between). Resetting is required the first time the function is called after rendering in another mode, after the glDrawElements() function is called, and after the library is initialized.<br>When rendering with either function, resetting is required per each render start command if either GL_TRIANGLE_STRIP or GL_TRIANGLE_FAN was passed to the rendering function. |
| KICK_COMMAND1<br>KICK_COMMAND2 | Special | When starting to render, write a 1 to any bit.<br>KICK_COMMAND1(0x022E) is glDrawArrays(), and KICK_COMMAND2(0x022F) is glDrawElements(). |

| | | |
|---|---|---|
| CLEAR_POST_VTX | Special | Immediately after starting to render, write a 1 to any bit. This must be set each time rendering is performed. |
| clearFrameBufferCacheData<br>Register 0x0111, bit [0:0] | 1 | Write a 1 immediately after the command to start rendering.<br>For more information about these registers, see 8.8.21. Framebuffer Cache Clear Setting Registers (0x0110, 0x0111). |
| samplerType0<br>samplerType1<br>samplerType2<br>Register 0x0080, bits [2:0] | 1<br>1<br>1 | Immediately before the command to start rendering, set to 1 for just those texture units that must be enabled, and then set to 0 immediately after the command to start rendering. It does not cause any problems in operation to leave this set to 1 for the texture units that must be enabled. However, setting this to 0 can help reduce power consumption, so it is set to 0 when not rendering.<br>For more information about these registers, see 8.8.6.2. Texture Sampler Type Setting Registers (0x0080, 0x0083). |
| DRAW_START | 1 | Set to 1 when the library is initialized. When this bit has a value of 1, rendering is not performed properly. When this bit has a value of 0, commands that set registers 0x02B0 through 0x02DF are not applied properly.<br>The glDrawElements() and glDrawArrays() functions generate commands to set this bit equal to 0 immediately before the render start command and return it to 1 immediately afterward.<br>It is acceptable to leave this bit set to 0 after the render start command, as long as you do not use any commands that set registers 0x02B0 through 0x02DF. |
| DUMMY_CMD | 8 | Two commands that each write a value of 0x0 here are required immediately after each render start command. Commands that write to this register at that time are dummy commands, so the setting values themselves have no meaning. |
| Register 0x02BA, bits [31:16] | 16 | Running a command that writes 0x7FFF to these bits after a render start command reduces power consumption.<br>Failure to do so does not cause any problems, but if you do write to these bits use a byte enable setting of 0xC so that you avoid writing to bits [15:0]. Also note that DRAW_START must be set to 1 before this register can be set. |
| Register 0x028A, bits [31:16] | 16 | Running a command that writes 0x7FFF to these bits after a render start command reduces power consumption.<br>Failure to do so does not cause any problems, but if you do write to these bits use a byte enable setting of 0xC so that you avoid writing to bits [15:0].<br>When you are not using geometry shaders (that is, when bit [0:0] of register 0x0244 and bits [1:0] of register 0x0229 are 0), commands to set this register are unnecessary because the values set in bits [31:16] of register 0x02BA double as this register's settings. |

### Notes About Command Dependencies

Bits [31:16] of register 0x02BA must be set after DRAW_START is set.

While CALL_DRAWARRAYS is set to 1, registers other than 0x0200 through 0x0254 and 0x0280 through 0x02DF might not be set properly. Set these registers only when CALL_DRAWARRAYS is

0. This does not apply to dummy commands that set bits [31:24] of register 0x025E, however.

There are several other commands that must set registers immediately after a render start command, but there are no dependencies affecting the order of those commands.

#### Note on the INDEX_ARRAY_OFFSET Setting

In the description of INDEX_ARRAYOFFSET in the table (bits [27:0] of register 0x0227), it states that either 0 or 0x20 must be set depending on conditions. More precisely speaking, if the setting is performed so that the following conditions are not met, values other than 0 and 0x20 will also be set correctly.

- If VERTEX_NUM is greater than 0x10:
  ((VERTEX_NUM – 0x10) × 2 + (ARRAY_BASE_ADDR << 4 ) +
  INDEX_ARRAY_OFFSET) & 0xFFF >= 0xFE0

- If VERTEX_NUM is less than or equal to 0x10:
  (ARRAY_BASE_ADDR << 4 + INDEX_ARRAY_OFFSET) & 0xFFF >= 0xFE0

The setting value for VERTEX_NUM is bits [31:0] of register 0x0228, and for ARRAY_BASE_ADDR, bits [28:1] of register 0x0200.

#### Limitations Specific to Load Arrays

When starting rendering for write operations to KICK_COMMAND2 (register 0x022F), the number of load arrays must be under 12. For more information, see Load Array Limitations.

## 8.8.19.2. Settings When Vertex Buffers Are Not Used

This section describes register settings when rendering without using vertex buffers by explaining how they differ from the register settings when vertex buffers are used. The bit layout is the same as in Figure 8-66. The names in the bit layout correspond to the following settings. Commands that set vertex attribute data are handled in the same way as render start commands. All these commands must be set before the render start command, unless indicated otherwise.

**Table 8-55. Names and Their Settings (Setting Registers When Vertex Buffers Are Used for Rendering)**

| Name | Bits | Description |
|---|---|---|
| DRAW_MODE<br>DRAW_TRIANGLES1<br>DRAW_TRIANGLES2 | 2<br>1<br>1 | Sets the rendering mode. This setting is specified by the *mode* parameter of the glDrawElements or glDrawArrays() function. This does not need to be reset unless settings change.<br>0x0: GL_TRIANGLES<br>0x1: GL_TRIANGLE_STRIP<br>0x2: GL_TRIANGLE_FAN<br>0x3: GL_GEOMETRY_PRIMITIVE_DMP<br>Both DRAW_TRIANGLES1 and DRAW_TRIANGLES2 are set to 0. |
| CALL_DRAWARRAYS | 1 | Calling either glDrawElements or glDrawArrays has the same effect. Because the initial setting is 0 when the library is initialized, set this to 1 when the function is called and reset to 0 when the function ends. |

| | | |
|---|---|---|
| | | While this is set to 1, registers other than 0x0200 through 0x0254 and 0x0280 through 0x02DF might not be set properly. |
| INDEX_ARRAY_OFFSET | 28 | Unused. |
| INDEX_ARRAY_TYPE | 1 | Unused. |
| VERTEX_NUM | 32 | Unused. The amount of input vertex attribute data determines the number of vertices to process. |
| FIRST_OFFSET | 32 | Unused. |
| RESET_VTX_INFO | 1 | When rendering with either the glDrawElements() or glDrawArrays() function and passing GL_TRIANGLES, this information does not need to be reset when making the second or subsequent calls as long as the number of vertices is a multiple of three. The glDrawElements() function (if called) does not use vertex buffers, and the function (either function) is called successively in the same mode. Resetting is required the first time the function is called after rendering in another mode, after a glDrawElements() function is called that uses vertex buffers, and after the library is initialized.<br>In other rendering modes (GL_GEOMETRY_PRIMITIVE_DMP, GL_TRIANGLE_STRIP, and GL_TRIANGLE_FAN), this is no different from when vertex buffers are used. |
| KICK_COMMAND1<br>KICK_COMMAND2 | Special | Do not set anything for these arguments. |
| CLEAR_POST_VTX | Special | No different from when vertex buffers are used. |
| clearFrameBufferCacheData register 0x0111 [0:0] | 1 | No different from when vertex buffers are used. |
| samplerType0<br>samplerType1<br>samplerType2<br>Register 0x0080, bits [2:0] | 1<br>1<br>1 | No different from when vertex buffers are used. |
| DRAW_START | 1 | No different from when vertex buffers are used. |
| Register 0x02BA, bits [31:16] | 16 | No different from when vertex buffers are used. |
| Register 0x028A, bits [31:16] | 16 | No different from when vertex buffers are used. |

The registers described in 8.8.1.8. Fixed Vertex Attribute Value Setting Registers (0x0232 – 0x0235) are used to input vertex attribute data.

First, 0xF is written to bits [3:0] of register 0x0232. Next, the vertex attribute data is converted into 24-bit floating-point numbers, and the resulting three 32-bit segments of data are written to registers 0x0233, 0x0234, and 0x0235, in that order. This 24-bit floating-point data written to the three, 32-bit segments is created in the same way as the data presented in the section titled 24-Bit Floating-Point Input Mode.

When vertex buffers are not used, none of the register settings described in 8.8.1.9. Vertex Attribute Array Setting Registers (0x0200 – 0x0227) are necessary. Dependencies that affect the order in which commands must be set are the same as when vertex buffers are used.

## 8.8.20. Geometry Shader Setting Registers (0x0280 – 0x02AF and Others)

There are multiple vertex shader processors built into the GPU for vertex operations. One of these processors serves as the geometry shader processor when geometry shaders are in use. This processor is called the shared processor. Because the shared processor operates as a vertex shader processor when geometry shaders are not in use, the resources for floating-point registers, Boolean registers, and other registers are set using vertex shader values. These values set as vertex shader settings must be changed to geometry shader settings when you switch from not using geometry shaders to using them. Similarly, the values set as geometry shader settings must be changed to vertex shader settings when you switch from using geometry shaders to not using them.

Registers 0x02B0 through 0x02DF are used to set the vertex shader processors. When you set any of the registers in this range, those settings are applied to all of the vertex shader processors. Generally these settings are also applied to the shared processor, except when bit [0:0] of register 0x0244 has a value of 1. In that case, vertex shader processor settings are not applied to the shared processor. Conversely, if the same bit has a value of 1 and bits [1:0] of register 0x0229 are 0x0, the settings are applied to the shared processor.

To set the shared processor only, make settings just as you would in registers 0x02B0 through 0x02DF, but first subtract 0x30 from the register addresses. In other words, use registers 0x0280 through 0x02AF to set only the shared processor.

Registers 0x0280 through 0x02AF have geometry shader–specific settings when geometry shaders are in use, but they must be set to the same settings as registers 0x02B0 through 0x02DF when geometry shaders are not in use. It is possible to meet this requirement by setting bit [0:0] of register 0x0244 equal to 0 and bits [1:0] of register 0x0229 equal to 0x0 (ensuring that vertex shader processor settings are also applied to the shared processor), and then resetting registers 0x02B0 through 0x02DF.

To use geometry shaders, you not only need to set these registers involved with the shared processor, but you must also set registers involved with other aspects, such as input and output.

## 8.8.20.1. Floating-Point Constant Registers (0x0290, 0x0291 – 0x0298)

The values to set in these registers are the same as those in 8.8.1.1. Floating-Point Constant Registers (0x02C0, 0x02C1 – 0x02C8). Only the register addresses are different.

Register 0x0290 corresponds to register 0x02C0 and registers 0x0291 through 0x0298 correspond to registers 0x02C1 through 0x02C8.

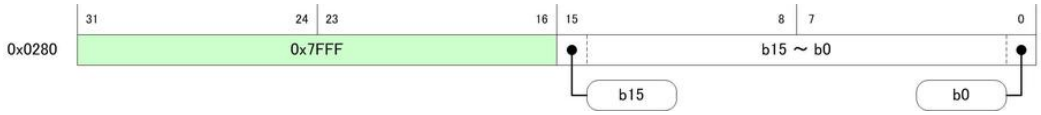
**Figure 8-67. Bit Layout for the Floating-Point Constant Register Index Specifier (0x0290)**

## 8.8.20.2. Boolean Register (0x0280)

The values to set in this register are the same as those in 8.8.1.2. Boolean Register (0x02B0). Only the register address is different.

Register 0x0280 corresponds to register 0x02B0.

**Figure 8-68. Bit Layout of Boolean Register (0x0280)**




## 8.8.20.3. Integer Registers (0x0281 – 0x0284)

The values to set in these registers are the same as those in 8.8.1.3. Integer Registers (0x02B1 – 0x02B4). Only the register addresses are different.

Registers `0x0281` through `0x0284` correspond to registers `0x02B1` through `0x02B4`.

**Figure 8-69. Bit Layout of Integer Registers (0x0281 - 0x0284)**

## 8.8.20.4. Program Code Setting Registers (0x028F, 0x029B – 0x02A3, 0x02A5 – 0x02AD)

The values to set in these registers are the same as those in 8.8.1.4. Program Code Setting Registers (0x02BF, 0x02CB – 0x02D3, 0x02D5 – 0x02DD). Only the register addresses are different.

Register `0x028F` corresponds to register `0x02BF`, registers `0x029B` through `0x02A3` correspond to registers `0x02CB` through `0x02D3`, and registers `0x02A5` through `0x02AD` correspond to registers `0x02D5` through `0x02DD`.

**Figure 8-70. Bit Layout of Program Code Loading Registers (0x029B – 0x02A3)**

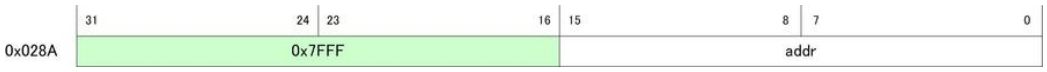
**Figure 8-71. Bit Layout of Swizzle Pattern Loading Registers (0x02A5 – 0x02AD)**

## 8.8.20.5. Starting Address Setting Register (0x028A)

The values to set in this register are the same as those in 8.8.1.5. Starting Address Setting Register (0x02BA). Only the register address is different.

Register `0x028A` corresponds to register `0x02BA`.

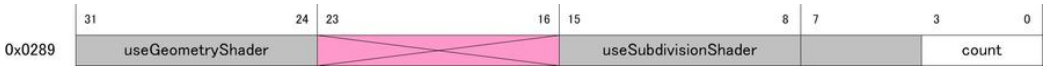
**Figure 8-72. Bit Layout of Starting Address Setting Register (0x028A)**




## 8.8.20.6. Vertex Attribute Input Count Setting Register (0x0289)

The register that sets the number of vertex attributes to input to the geometry shader has the following layout.

**Figure 8-73. Bit Layout of the Vertex Attribute Input Count Setting Register (0x0289)**




Set the value of `count` to the number of vertex attributes to input minus 1. The same number of vertex attributes that are output from the vertex shader (including `generic` attributes) are input to the geometry shader.

The number of vertex attributes set here is not the number of `#pragma output_map` definitions in vertex shader assembly, it is the number of output registers defined by `#pragma output_map`. In other words, even if multiple `#pragma output_map` definitions are used for the individual components of a single output register, they still count as a single register.

## 8.8.20.7. Input Register Mapping Setting Registers (0x028B, 0x028C)

The values set in these registers are the same as those in 8.8.1.7. Input Register Mapping Setting Registers (0x02BB, 0x02BC). Only the register addresses are different. However, when a geometry shader such as the line shader is used, set register `0x028B` equal to `0x76543210` and set register `0x028C` equal to `0xFEDCBA98`.

Registers `0x028B` through `0x028C` correspond to registers `0x02BB` through `0x02BC`.

**Figure 8-74. Bit Layout of Input Register Mapping Setting Registers (0x028B, 0x028C)**

## 8.8.20.8. Output Register Use Count Setting Registers (0x004F, 0x025E)

The number of output registers used by geometry shaders is configured by some of the same registers described in 8.8.1.10. Output Register Use Count Setting Registers (0x004F, 0x024A, 0x0251, 0x025E).

Set the unchanged raw number of output registers to use in *count1* (register 0x004F), and set the number of used output registers minus 1 in *count2* (register 0x025E only). Note that these register settings have different values and bit widths.

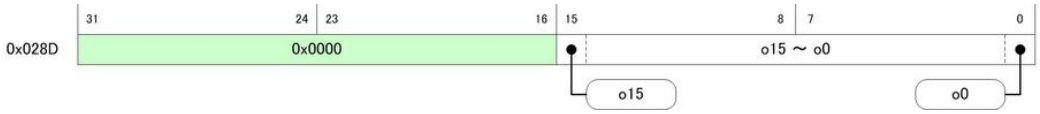
The number of output registers is the number of output registers defined by `#pragma output_map` for the geometry shader. In other words, even if multiple `#pragma output_map` definitions are used for the individual components of a single output register, they still count as a single register.

## 8.8.20.9. Output Register Mask Setting Register (0x028D)

The values to set in this register are the same as those in 8.8.1.11. Output Register Mask Setting Register (0x02BD). Only the register address is different.

Register 0x028D corresponds to register 0x02BD.

**Figure 8-75. Bit Layout of Output Register Mask Setting Register (0x028D)**




## 8.8.20.10. Output Register Attribute Setting Registers (0x0050 – 0x0056, 0x0064)

The values to set in these registers are the same as described in 8.8.1.12. Output Register Attribute Setting Registers (0x0050 – 0x0056, 0x0064), except for the fact that they modify the vertex attributes output by geometry shaders instead of those output by vertex shaders.

Geometry shader output attributes are determined by the `#pragma output_map` settings defined in the geometry shader assembly code. This information is generated in the map files output by the shader assembly code linker. Several geometry shaders define `generic` attributes as output attributes. Output attributes defined as `generic` attributes take the `#pragma output_map` settings that are defined only by the linked vertex shader. When this is done, the `generic` attributes defined by the vertex shader are excluded.

**Note:** For more information about map files, see the Vertex Shader Reference.

## 8.8.20.11. Output Attribute Clock Control Register (0x006F)

The values to set in this register are the same as those in 8.8.1.13. Output Attribute Clock Control Register (0x006F), except for the fact that they modify geometry shaders instead of vertex shaders.

## 8.8.20.12. Other Setting Registers (0x0229, 0x0252, 0x0254, 0x0289)

The following table shows other registers to set when geometry shaders are used.

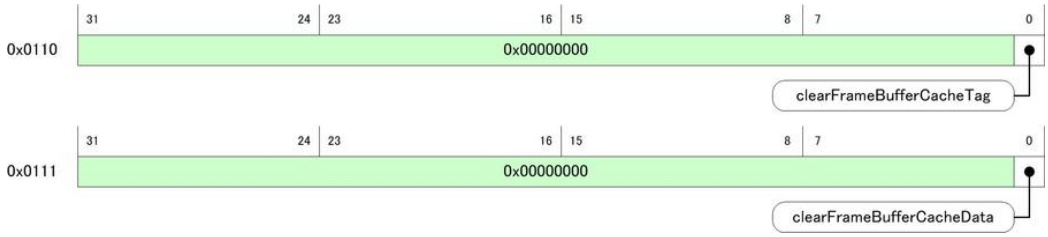
**Table 8-56. Other Registers (Geometry Shaders)**

| Register | Description |
|---|---|
| 0x0229, bit [1:0] | Set to `0x2` when geometry shaders are used, and set to `0x0` when geometry shaders are not used.<br>Dummy commands are required before and after any command that sets this register. Use dummy commands with a byte enable setting of `0`. Ten (10) dummy commands that set register `0x0251` and thirty (30) dummy commands that set register `0x0200` are required immediately before a command that sets this register. Thirty (30) dummy commands that set register `0x0200` are required immediately after a command that sets this register. Note that dummy commands are not required, however, when a command sets any bit in this register other than the bits mentioned in this row. |
| 0x0229, bits [31:31] | Set to `0x1` when subdivision shaders (Loop or Catmull-Clark) are used, and set to `0x0` when other geometry shaders are used or when geometry shaders are not used. |
| 0x0252, bits [31:0] | Set to `0x00000001` when subdivision shaders (Loop or Catmull-Clark) are used, `0x01004302` when particle systems are used, and `0x00000000` when other geometry shaders are used or when geometry shaders are not used. |
| 0x0289, bits [31:24] | Set to `0x08` when geometry shaders are used. Set to `0xA0` when geometry shaders are not used. |
| 0x0289, bits [15:8] | Set to `0x01` when subdivision shaders (Loop or Catmull-Clark) or particle systems are used, and `0x00` when other geometry shaders are used or when geometry shaders are not used. |
| 0x0254, bits [4:0] | Set to `0x3` when Catmull-Clark subdivision shaders are used and `0x2` when Loop subdivision shaders are used. In all other cases these bits are unused. |

# 8.8.21. Framebuffer Cache Clear Setting Registers (0x0110, 0x0111)

The registers that handle settings for clearing the framebuffer cache have the following layout.

**Figure 8-76. Bit Layout of Framebuffer Cache Clear Setting Registers (0x0110, 0x0111)**




When a value of 1 is written to `clearFrameBufferCacheData`, cache data is flushed for both the color and depth buffers. When a value of 1 is written to `clearFrameBufferCacheTag`, cache tags are cleared for both the color and depth buffers. You must use the command that clears the cache tags immediately after you use the command that flushes the cache data. In other words, the cache must first be flushed, and then the cache tags must be cleared.

Commands are generated to set these registers when the `glFlush`, `glFinish`, or `glClear()` function is called, when the state flag `NN_GX_STATE_FBACCESS` is validated, and when the state

flag NN_GX_STATE_FRAMEBUFFER is validated due to changes in the color or depth buffer addresses. Also, when the 3D Command Buffer is split by the nngxSplitDrawCmdlist() function or other means, these commands are inserted immediately before the command that issues interrupts.

A standalone command to flush cache data is generated immediately after the render start command caused by the glDrawArrays or glDrawElements() function.

Both of these registers must be set when any of the following have happened: all rendering is complete (and the rendering results have not been accessed yet), the color or depth buffer has been cleared, either buffer's settings (size, address, or format) have changed, or the access pattern has changed.

Although you must normally generate a command to set clearFrameBufferCacheData immediately after each render command, this is not necessary when the following two conditions are met.

- After you have generated a command to set clearFrameBufferCacheData immediately following a render command, you do not generate any commands that set registers 0x0100 through 0x0130 until the next render command.
- Before you set data in registers 0x0080 through 0x00B7 following a render command, you generate three dummy commands (whose data and byte enable are both 0) to register 0x0080.

To set data in registers 0x0100 through 0x0130 after a render command, you must first generate a single command that sets clearFrameBufferCacheData. As long as you have generated at least one such command, you can safely generate any number of commands that set data in registers 0x0100 through 0x0130 before the next render command.

In the same way, as long as you have generated three dummy commands (whose data and byte enable are both 0) to register 0x0080 after a render command, you can safely generate any number of commands that set registers 0x0080 through 0x00B7 before the next render command.

## 8.8.22. Split Command Setting Register (0x0010)

Writing a value of 0x12345678 to register 0x0010 causes a GPU interrupt to occur. Set this command when splitting the 3D Command Buffer.

## 8.8.23. Command Buffer Execution Registers (0x0238 – 0x023D)

Command buffers are usually executed based on the information in queued render command requests. By using the command buffer execution registers, you can execute a different command buffer with the 3D commands accumulated in the first command buffer without needing to issue a GPU interrupt via a split command.

**Figure 8-77. Command Buffer Execution Registers (0x0238 – 0x023D)**

Two channels have been prepared for the command buffer execution interface. Channel 0 is used for normal command buffer execution. You cannot execute both channels simultaneously.

To execute a command buffer, you need to set three registers: one for the address of the command buffer to execute, one for the size of said command buffer, and one that is the execution register (also known as the kick register).

In the register that sets the address of the command buffer you want to execute (`BUFFER_ADDR_CHANNEL0` or `BUFFER_ADDR_CHANNEL1`), specify a value that is the result of dividing the command buffer's physical address by 8 (this value is the 8-byte address). You must specify an even number.

In the register that sets the address of the command buffer you want to execute (`BUFFER_ADDR_CHANNEL0` or `BUFFER_ADDR_CHANNEL1`), specify a value that is the result of dividing the command buffer's physical address by 8 (this value is the 8-byte address). You must specify an even number.

When any value is written to the kick register (`KICK_CHANNEL0` or `KICK_CHANNEL1`), the command buffer is executed using the address and size settings for that channel. Any value may be written. If the byte enable is not `0`, the command buffer is executed, regardless of what data is written to the kick register. When writing to the kick register with a command buffer 3D command, place that command at the end of the command buffer. The command buffer address and size must both be 16-byte aligned (the values divided by 8 must not be odd), so the last address after storing a kick command (a command that writes to `KICK_CHANNEL0` or `KICK_CHANNEL1`) must be 16-byte aligned. The address and size settings do not affect the currently executing command buffer until a kick command is executed.

**Notes**

Pay attention to the following conditions when you use settings in the command buffer execution registers.

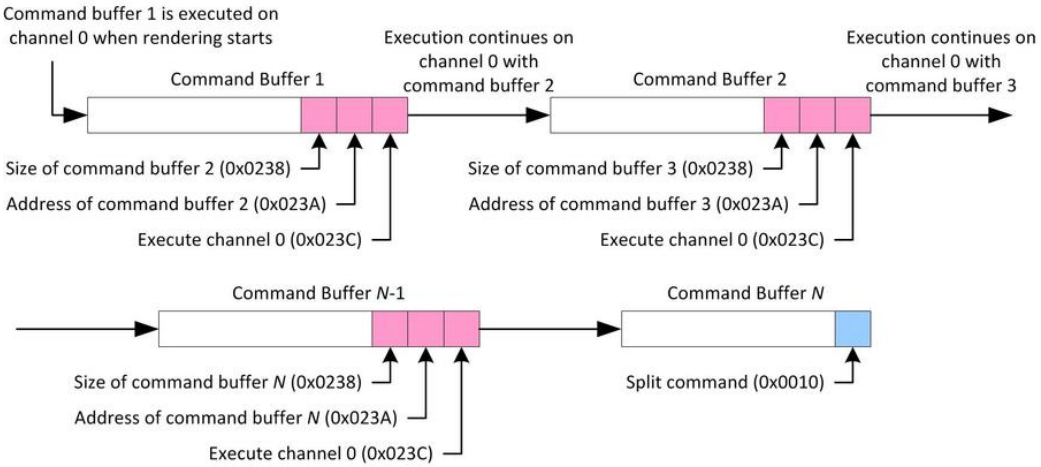
- Kick commands must be placed at the end of the command buffer. Specify the command buffer size so that the kick command is last.
- You cannot write to the kick register in the middle of a burst access. If you write to the kick register at the end of the burst access, however, a kick command will be executed.
- Although the address and size settings for the two channels are maintained after a kick command is executed, the settings for channel 0 are overwritten when `nngx()` functions or other functions execute 3D commands.

- The command buffer's processing might not run correctly if you have not flushed the command buffer region you want to execute.
- Channel 0 and channel 1 cannot be executed simultaneously.
- Command execution is interrupted for the time between executing a kick command and executing the specified command in the command buffer. Consequently, frequently executing kick commands that specify a small command buffer can increase the GPU processing load by a noticeable degree.

## 8.8.23.1. Consecutive Execution of Command Buffers

As shown in the following figure, you can execute *N* command buffers in a row by storing commands that write to the command buffer execution registers at the end of your command buffers.

**Figure 8-78. Consecutive Execution of Command Buffers**


Command buffer 1 is executed on channel 0 when rendering starts
Command Buffer 1
Execution continues on channel 0 with command buffer 2
Command Buffer 2
Execution continues on channel 0 with command buffer 3
Size of command buffer 2 (0x0238)
Address of command buffer 2 (0x023A)
Execute channel 0 (0x023C)
Size of command buffer 3 (0x0238)
Address of command buffer 3 (0x023A)
Execute channel 0 (0x023C)
Command Buffer N-1
Command Buffer N
Size of command buffer N (0x0238)
Address of command buffer N (0x023A)
Execute channel 0 (0x023C)
Split command (0x0010)

A split command is normally stored at the end of the command buffer, as shown in command buffer *N*. By replacing this split command with three commands (one to set the size of the command buffer to execute next, one to set the address of said buffer, and one kick command), you can execute multiple command buffers without ever issuing a GPU interrupt, reducing the CPU load. However, a split command must be the last command that is executed in the last command buffer.

To execute the first command buffer, call the `nngxAdd3DCommand()` function with the command buffer's address and size specified as the *`bufferaddr`* and *`buffersize`* parameters, respectively, and `GL_FALSE` specified as the *`copycmd`* parameter.

## 8.8.23.2. Repeated Execution of the Same Command Buffer

The example in the previous section requires you to provide a separate command buffer, even if you just want to repeatedly execute the same commands. By using the two available channels, however, you can repeatedly execute the same command buffer.

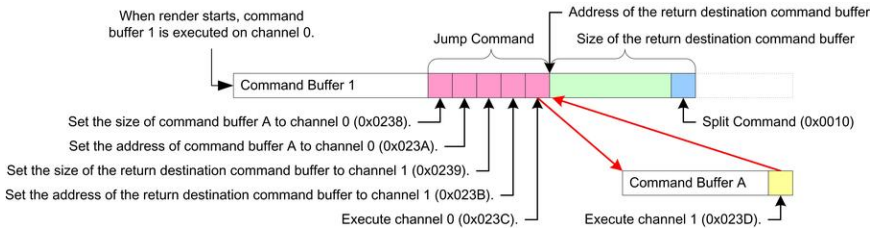
For example, assume that you write to the combination of registers listed in Table 8-61. These registers are all consecutive, so you can formulate a single write command that writes to all of them at the same time: a jump command. But if you use a jump command, the command buffer

that you jump to must end with a kick command for channel 1.

**Table 8-57. Structure of a Jump Command**

| Register | Value to Set |
|---|---|
| BUFFER_SIZE_CHANNEL0 | Size of the command buffer to jump to. |
| BUFFER_SIZE_CHANNEL1 | Size of the command buffer to return to (until the next jump command or split command). |
| BUFFER_ADDR_CHANNEL0 | Address of the command buffer to jump to. |
| BUFFER_ADDR_CHANNEL1 | Address of the command buffer to return to (this address must be located immediately after the jump command described by this table). |
| KICK_CHANNEL0 | Any value (executes a kick command). |

**Figure 8-79. Structure of a Jump Command**

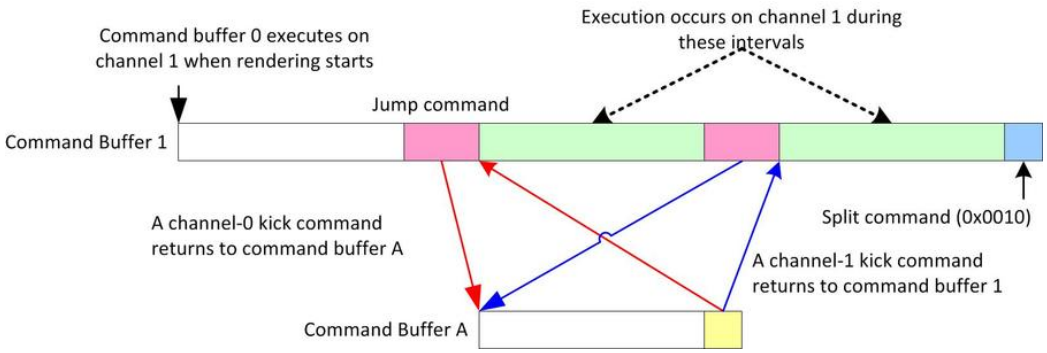

When render starts, command buffer 1 is executed on channel 0.
Jump Command
Address of the return destination command buffer
Size of the return destination command buffer
Command Buffer 1
Set the size of command buffer A to channel 0 (0x0238).
Set the address of command buffer A to channel 0 (0x023A).
Set the size of the return destination command buffer to channel 1 (0x0239).
Set the address of the return destination command buffer to channel 1 (0x023B).
Execute channel 0 (0x023C).
Split Command (0x0010)
Command Buffer A
Execute channel 1 (0x023D).

In this method, the address information for the command buffer to return to is set ahead of time in the jump source command buffer. There is no need to include the address information in the jump destination command buffer.

In the figure, command buffer 1 configures the jump destination to channel 0 and the return destination to channel 1. After these settings have been configured, a channel-0 kick command executes command buffer A, and then a channel-1 kick command at the end of command buffer A returns execution to command buffer 1. Note that you must firmly decide ahead of time which channel you will use when returning from the jump because you cannot change the content of command buffer A that you jump to while you are executing it.

If you place a jump command at the end of the return destination, you can execute the same command buffer repeatedly. To execute the first command buffer, call the `nngxAdd3DCommand()` function with the command buffer's address and size up to the first kick command specified in *`bufferaddr`* and *`buffersize`*, respectively, and GL_FALSE specified as the *`copycmd`* parameter. Set the size from the return destination address to the jump source to the next kick command as the command size for channel 1 included in each jump command.

**Figure 8-80. Repeated Execution of the Same Command Buffer**


Command buffer 0 executes on channel 1 when rendering starts
Execution occurs on channel 1 during these intervals
Jump command
Command Buffer 1
Split command (0x0010)
A channel-0 kick command returns to command buffer A
A channel-1 kick command returns to command buffer 1
Command Buffer A

This method allows you to perform operations such as the following.

- Repeatedly execute a specific rendering process.
- Edit information stored at the destination of the jump, and thereby branch the rendering process.
- Modularize the rendering process, and construct a scene with jump commands alone.

## 8.8.24. Settings for Undocumented Bits

Some of the registers described so far have undocumented bits. You must use a byte enable setting of `0` to avoid accessing some of these undocumented bits, and you must set others to fixed values. This section provides information about access to those bits. Although bits that are completely undocumented (mentioned neither in the preceding sections nor this section) can, in theory, be set to any value without affecting the GPU, we recommend that you use a byte enable setting of `0` whenever possible. Do not set any registers that are not documented.

Applications must not issue commands to initialize registers in which the following table instructs you to set a fixed value because `nngxInitialize()` initializes the registers with these values. If any byte in a single register contains both bits that must be set to fixed values and bits that can be changed, both types of values must be written at the same time.

**Table 8-58. Settings for Undocumented Bits**

| Register | Description |
|---|---|
| 0x0061, bits [31:8] | Set a byte enable of `0` to ensure no access. |
| 0x0062, bits [31:8] | Set a byte enable of `0` to ensure no access. |
| 0x006A, bits [31:24] | Set a byte enable of `0` to ensure no access. |
| 0x006E, bits [24:24] | Set to `1`. |
| 0x0080, bits [3:3] and bits [31:24] | Set to `0`. |
| 0x0080, bits [12:12] | Set to `1`. |
| 0x0080, bits [23:17] | Set to `0` when you write to bit [`16:16`] of this register to clear the texture cache. Otherwise, set a byte enable of `0` to ensure no access. |
| 0x0083, bits [17:16] | Set to `0`. |
| 0x0093, bits [17:16] | Set to `0`. |
| 0x009B, bits [17:16] | Set to `0`. |
| 0x00AC, bits [10:3] | Set to `0x60`. |
| 0x00AD, bits [31:8] | Set to `0xE0C080`. |
| 0x00E0, bits [25:24] | Set to `0`. |
| 0x0100, bits [25:16] | Set to `0x0E4`. |
| 0x0110, bits [31:1] | Set to `0`. |
| 0x0111, bits [31:1] | Set to `0`. |
| 0x011E, bits | |

| | |
|---|---|
| [24:24] | Set to 1. |
| 0x01C0, bits [9:8] | Set to 0. |
| 0x01C0, bits [19:18] | Set to 0. |
| 0x01C0, bits [29:28] | Set to 0. |
| 0x01C3, bits [11:8] | Set to 0x4. |
| 0x01C3, bits [31:31] | Set to 1. |
| 0x01C4, bits [18:18] | Set to 1. |
| 0x0229, bits [9:9] | Set to 0. |
| 0x0229, bits [23:16] | Set a byte enable of 0 to ensure no access. |
| 0x0244, bits [31:8] | Set a byte enable of 0 to ensure no access. |
| 0x0245, bits [7:1] | Set to 0. |
| 0x0245, bits [31:8] | Set a byte enable of 0 to ensure no access. |
| 0x0253, bits [31:16] | Set a byte enable of 0 to ensure no access. |
| 0x025E, bits [16:16] | Set a byte enable of 0 to ensure no access. |
| 0x025F, bits [31:1] | Set to 0. |
| 0x0280, bits [31:16] | Set to 0x7FFF. |
| 0x0289, bits [23:16] | Set a byte enable of 0 to ensure no access. |
| 0x028A, bits [31:16] | Set to 0x7FFF. |
| 0x028D, bits [31:16] | Set to 0x0000. |
| 0x02B0, bits [31:16] | Set to 0x7FFF. |
| 0x02B9, bits [15:8] | Set to 0x00. |
| 0x02B9, bits [23:16] | Set a byte enable of 0 to ensure no access. |
| 0x02B9, bits [31:24] | Set to 0xA0. |
| 0x02BA, bits [31:16] | Set to 0x7FFF. |
| 0x02BD, bits [31:16] | Set to 0x0000. |

## 8.8.25. Register Settings When Geometry Shaders Are Used

This section explains which values to set in the registers described by 8.8.20. Geometry Shader Setting Registers (0x0280 – 0x02AF and Others) when you use the geometry shaders provided by the SDK.

## 8.8.25.1. Point Shaders

The following table shows the register values to set when using point shaders.

**Table 8-59. Register Settings When Using Point Shaders**

| Register | Description |
|---|---|
| 0x004F, bits [2:0] | Set to the number of output registers defined by the linked vertex shader. This count does not include `generic` attributes. |
| 0x0050 – 0x0056 | Set these, starting from `0x0050`, by packing them with the output register attributes defined by the linked vertex shader.<br>Set registers `0x0050` through `0x03020100`. The point size is output as a `generic` attribute, and it does not affect these registers. Continuing from `0x0051`, pack the registers with the defined attributes in order of output register index, starting from the smallest index.<br>For example, because a point sprite's vertex coordinates are assumed to be followed by texture coordinates, for attributes defined by `#pragma output_map(texture0, o2.xy)` you would set register `0x0051` to `0x1F1F0D0C`. If any attributes are unused, that portion is packed bytewise with `0x1F`.<br>If any attributes are unused, that portion is packed bytewise with `0x1F`. |
| 0x0064 | Set in accordance with the output register attributes defined by the linked vertex shader. |
| 0x006F | Set in accordance with the output register attributes defined by the linked vertex shader. |
| 0x0229, bits [31:31] | Set to `0`. |
| 0x0242, bits [3:0] | Set to one less than the number of vertex attributes to input to the linked vertex shader. |
| 0x024A, bits [3:0] | Set to one less than the number of output registers defined by the linked vertex shader. This count includes generic attributes. |
| 0x0251, bits [3:0] | Set to one less than the number of output registers defined by the linked vertex shader. This count includes generic attributes. |
| 0x0252 | Set to `0x00000000`. |
| 0x0254, bits [4:0] | No setting required. |
| 0x025E, bits [3:0] | Set to one less than the number of output registers defined by the linked vertex shader. This count does not include `generic` attributes. |
| 0x0280, bits [15:0] | Set to `0x0000`. |
| 0x0281, bits [23:0] | No setting required. |
| 0x0282, bits [23:0] | No setting required. |
| 0x0283, bits [23:0] | No setting required. |
| 0x0284, | |

| | |
|---|---|
| bits [23:0] | No setting required. |
| 0x0289, bits [3:0] | Set to one less than the number of output registers defined by the linked vertex shader. This count includes generic attributes. |
| 0x0289, bits [15:8] | Set to `0x00`. |
| 0x0289, bits [31:24] | Set to `0x08`. |
| 0x028D, bits [15:0] | Set to ((1 << *N*) - 1), where *N* is the number of output registers defined by the linked vertex shader. This count does not include `generic` attributes. |
| 0x0290 – 0x0293 | Write the following combinations to registers {`0x0290`, `0x0291`, `0x0292`, `0x0293`}, respectively. These are used to set floating-point constants. `{0x0000004C, 0x00000000, 0x00003F00, 0x00000000}` |

When you use point shaders, you must also set the reserved uniforms that are assigned to the following specific registers.

**Table 8-60. Point Shader Uniforms and Their Registers**

| Reserved Uniform | Allocated Register |
|---|---|
| dmp_Point.viewport | `c67.xy` |
| dmp_Point.distanceAttenuation | `b0` |

## 8.8.25.2. Line Shaders

The following table shows the register values to set when line shaders are used.

**Table 8-61. Register Settings When Using Line Shaders**

| Register | Description |
|---|---|
| 0x004F, bits [2:0] | Set to the number of output registers defined by the linked vertex shader. |
| 0x0050 – 0x0056 | Set these, starting from `0x0050`, by packing them with the output register attributes defined by the linked vertex shader.<br>Set registers `0x0050` through `0x03020100`. Continuing from `0x0051`, pack the registers with the defined attributes in order of output register index, starting from the smallest index.<br>If any attributes are unused, that portion is packed bytewise with `0x1F`. |
| 0x0064 | Set in accordance with the output register attributes defined by the linked vertex shader. |
| 0x006F | Set in accordance with the output register attributes defined by the linked vertex shader. |
| 0x0229, bits [31:31] | Set to `0`. |
| 0x0242, bits [3:0] | Set to one less than the number of vertex attributes to input to the linked vertex shader. |

| | |
|---|---|
| 0x024A, bits [3:0] | Set to one less than the number of output registers defined by the linked vertex shader. |
| 0x0251, bits [3:0] | Set to one less than the number of output registers defined by the linked vertex shader. |
| 0x0252 | Set to `0x00000000`. |
| 0x0254, bits [4:0] | No setting required. |
| 0x025E, bits [3:0] | Set to one less than the number of output registers defined by the linked vertex shader. |
| 0x0280, bits [15:0] | Set to `0x0000`. Bit [`15:15`] must be set each time rendering is performed. |
| 0x0281, bits [23:0] | No setting required. |
| 0x0282, bits [23:0] | No setting required. |
| 0x0283, bits [23:0] | No setting required. |
| 0x0284, bits [23:0] | No setting required. |
| 0x0289, bits [3:0] | Set to one less than the number of output registers defined by the linked vertex shader. |
| 0x0289, bits [15:8] | Set to `0x00`. |
| 0x0289, bits [31:24] | Set to `0x08`. |
| 0x028D, bits [15:0] | Set to ((1 << *N*) - 1), where *N* is the number of output registers defined by the linked vertex shader. |
| 0x0290 – 0x0293 | Write the following combinations to registers {`0x0290`, `0x0291`, `0x0292`, `0x0293`}, respectively. These are used to set floating-point constants. `{0x00000057, 0x40800040, 0x00003F00, 0x00000000}` |

When you use line shaders, you must also set the reserved uniforms that are assigned to the following specific registers.

**Table 8-62. Point Shader Uniforms and Their Registers**

| Reserved Uniform | Allocated Register |
|---|---|
| dmp_Line.width | `c67.xyzw` |

## 8.8.25.3. Silhouette Shaders

The following table shows the register values to set when using particle system shaders.

**Table 8-63. Register Settings When Using Silhouette Shaders**

| Register | Description |
|---|---|
| 0x004F, bits [2:0] | Set to 0x2. |
| 0x0050 – 0x0056 | Set register 0x0050 to 0x03020100. Set register 0x0051 to 0x0B0A0908. Set registers 0x0052 through 0x0056 to 0x1F1F1F1F. |
| 0x0064 | Set to 0x00000000. |
| 0x006F | Set to 0x00000003. |
| 0x0229, bits [31:31] | Set to 0. |
| 0x0242, bits [3:0] | Set to one less than the number of vertex attributes to input to the linked vertex shader. |
| 0x024A, bits [3:0] | Set to 0x2. |
| 0x0251, bits [3:0] | Set to 0x2. |
| 0x0252 | Set to 0x00000000. |
| 0x0254, bits [4:0] | No setting required. |
| 0x025E, bits [3:0] | Set to 0x1. |
| 0x0280, bits [15:0] | Set to 0x0000. Bit [15:15] must be set each time rendering is performed. |
| 0x0281, bits [23:0] | No setting required. |
| 0x0282, bits [23:0] | No setting required. |
| 0x0283, bits [23:0] | No setting required. |
| 0x0284, bits [23:0] | No setting required. |
| 0x0289, bits [3:0] | Set to 0x2 because the vertex shader outputs three vertex attributes: vertex coordinates, colors, and normals. |
| 0x0289, bits [15:8] | Set to 0x00. |
| 0x0289, bits [31:24] | Set to 0x08. |
| 0x028D, bits [15:0] | Set to 0x0003. |
| 0x0290 – 0x0293 | Write the following combinations to registers {0x0290, 0x0291, 0x0292, 0x0293}, respectively. These are used to set floating-point constants.<br>{0x0000004C, 0x40800040, 0x00003F00, 0x00000000}<br>{0x0000004D, 0x00000000, 0x00004140, 0x00410000} |

When you use silhouette shaders, you must also set the reserved uniforms that are assigned to the following specific registers.

**Table 8-64. Silhouette Shader Uniforms and Their Registers**

| Reserved Uniform | Allocated Register |
|---|---|
| dmp_Silhouette.width | c71.xy |
| dmp_Silhouette.openEdgeDepthBias | c71.z |
| dmp_Silhouette.color | c72.xyzw |
| dmp_Silhouette.openEdgeColor | c73.xyzw |
| dmp_Silhouette.openEdgeWidth | c74.xyzw |
| dmp_Silhouette.acceptEmptyTriangles | b0 |
| dmp_Silhouette.scaleByW | b1 |
| dmp_Silhouette.frontFaceCCW | b2 |
| dmp_Silhouette.openEdgeWidthScaleByW | b3 |
| dmp_Silhouette.openEdgeDepthBiasScaleByW | b4 |

## 8.8.25.4. Catmull-Clark Subdivision Shaders

The following table shows the register values to set when using Catmull-Clark subdivision shaders.

**Table 8-65. Register Settings When Using Catmull-Clark Subdivision Shaders**

| Register | Description |
|---|---|
| 0x004F, bits [2:0] | Set to the number of output registers defined by the linked vertex shader. |
| 0x0050 – 0x0056 | Set these, starting from 0x0050, by packing them with the output register attributes defined by the linked vertex shader.<br>Set registers 0x0050 through 0x03020100. Continuing from 0x0051, pack the registers with the defined attributes in order of output register index, starting from the smallest index.<br>If any attributes are unused, that portion is packed bytewise with 0x1F. |
| 0x0064 | Set in accordance with the output register attributes defined by the linked vertex shader. |
| 0x006F | Set in accordance with the output register attributes defined by the linked vertex shader. |
| 0x0229, bits [31:31] | Set to 1. |
| 0x0242, bits [3:0] | Set to one less than the number of vertex attributes to input to the linked vertex shader. |
| 0x024A, bits [3:0] | Set to one less than the number of output registers defined by the linked vertex shader. |
| 0x0251, bits [3:0] | Set to one less than the number of output registers defined by the linked vertex shader. |
| 0x0252 | Set to 0x00000001. |
| 0x0254, bits [4:0] | Set to 0x03. |
| | |

| | |
|---|---|
| 0x025E, bits [3:0] | Set to one less than the number of output registers defined by the linked vertex shader. |
| 0x0280, bits [15:0] | Set to `0x0000`. Bit [15:15] must be set each time rendering is performed. |
| 0x0281, bits [23:0] | No setting required. |
| 0x0282, bits [23:0] | The value to set varies with the linked geometry shader's program object file.<br>Set `0x0212FF` for `DMP_subdivision1.obj`.<br>Set `0x0216FF` for `DMP_subdivision2.obj`.<br>Set `0x021AFF` for `DMP_subdivision3.obj`.<br>Set `0x021EFF` for `DMP_subdivision4.obj`.<br>Set `0x0222FF` for `DMP_subdivision5.obj`.<br>Set `0x0226FF` for `DMP_subdivision6.obj`. |
| 0x0283, bits [23:0] | No setting required. |
| 0x0284, bits [23:0] | No setting required. |
| 0x0289, bits [3:0] | Set to one less than the number of output registers defined by the linked vertex shader. |
| 0x0289, bits [15:8] | Set to 0x01. |
| 0x0289, bits [31:24] | Set to `0x08`. |
| 0x028D, bits [15:0] | Set to ((1 << *N*) - 1), where *N* is the number of output registers defined by the linked vertex shader. |
| 0x0290 – 0x0293 | Write the following combinations to registers {`0x0290`, `0x0291`, `0x0292`, `0x0293`}, respectively. These are used to set floating-point constants.<br>`{0x0000004C, 0x3C80003B, 0x00003C80, 0x003E2000}`<br>`{0x0000004D, 0x0000003E, 0x00003C00, 0x003D8000}`<br>`{0x0000004E, 0x4300003D, 0x00003E80, 0x00420000}`<br>`{0x0000004F, 0x3C60003C, 0xC8003780, 0x00390000}`<br>`{0x00000050, 0x3D0C0039, 0x80003700, 0x003B8000}`<br>`{0x00000051, 0x3CC0003C, 0x70003A60, 0x003C2800}`<br>`{0x00000052, 0x3D16003B, 0x0C003500, 0x003D8000}`<br>`{0x00000053, 0x3DAAAA39, 0xC71C3C55, 0x55BE2AAA}`<br>`{0x00000054, 0x3D871C3A, 0x425E3C55, 0x55BE3C71}`<br>`{0x00000055, 0x3E200039, 0x00003B80, 0x00BDC000}`<br>`{0x00000056, 0x3D940039, 0x8FFF3C04, 0x00BE3600}`<br>`{0x00000057, 0x0000003F, 0x00004180, 0x00C0C000}`<br>`{0x00000058, 0x00000040, 0x00004230, 0x00C17000}`<br>`{0x00000059, 0x000000C0, 0xC000C350, 0x00428800}`<br>Set `{0x0000004B, 0x42000041, 0x80004100, 0x00400000}` only for `DMP_subdivision1.obj`.<br>Set `{0x0000004B, 0x42800042, 0x20004180, 0x00408000}` only for `DMP_subdivision2.obj`.<br>Set `{0x0000004B, 0x43000042, 0x80004200, 0x00410000}` only for `DMP_subdivision3.obj`.<br>Set `{0x0000004B, 0x43400042, 0xE0004240, 0x00414000}` only for `DMP_subdivision4.obj`.<br>Set `{0x0000004B, 0x43800043, 0x20004280, 0x00418000}` only for `DMP_subdivision5.obj`.<br>Set `{0x0000004B, 0x43C00043, 0x500042C0, 0x0041C000}` only for |

| | |
|---|---|
| | DMP_subdivision6.obj. |

When you use Catmull-Clark subdivision shaders, you must also set the reserved uniforms that are assigned to the following specific registers.

**Table 8-66. Catmull-Clark Subdivision Shader Uniforms and Their Registers**

| Reserved Uniform | Allocated Register |
|---|---|
| dmp_Subdivision.level | c74.x |
| dmp_Subdivision.fragmentLightingEnabled | b2 |

## 8.8.25.5. Loop Subdivision Shader

The following table shows the register values to set when Loop subdivision shaders are used.

**Table 8-67. Register Settings When Using Loop Subdivision Shaders**

| Register | Description |
|---|---|
| 0x004F, bits [2:0] | Set to the number of output registers defined by the linked vertex shader. This count does not include generic attributes. |
| 0x0050 – 0x0056 | Set these, starting from 0x0050, by packing them with the output register attributes defined by the linked vertex shader.<br>Set registers 0x0050 through 0x03020100. Continuing from 0x0051, pack the registers with the defined attributes in order of output register index, starting from the smallest index. Any generic attributes are ignored.<br>If any attributes are unused, that portion is packed bytewise with 0x1F. |
| 0x0064 | Set in accordance with the output register attributes defined by the linked vertex shader. |
| 0x006F | Set in accordance with the output register attributes defined by the linked vertex shader. |
| 0x0229, bits [31:31] | Set to 1. |
| 0x0242, bits [3:0] | Set to one less than the number of vertex attributes to input to the linked vertex shader. |
| 0x024A, bits [3:0] | Set to one less than the number of output registers defined by the linked vertex shader. This count includes generic attributes. |
| 0x0251, bits [3:0] | Set to one less than the number of output registers defined by the linked vertex shader. This count includes generic attributes. |
| 0x0252 | Set to 0x00000001. |
| 0x0254, bits [4:0] | Set to 0x02. |
| 0x025E, bits [3:0] | Set to one less than the number of output registers defined by the linked vertex shader. This count does not include generic attributes. |
| 0x0280, bits [15:0] | Set to 0x0000. Bit [15:15] must be set each time rendering is performed. |

| | |
|---|---|
| 0x0281, bits [23:0] | No setting required. |
| 0x0282, bits [23:0] | No setting required. |
| 0x0283, bits [23:0] | No setting required. |
| 0x0284, bits [23:0] | No setting required. |
| 0x0289, bits [3:0] | Set to one less than the number of output registers defined by the linked vertex shader. This count includes generic attributes. |
| 0x0289, bits [15:8] | Set to 0x01. |
| 0x0289, bits [31:24] | Set to 0x08. |
| 0x028D, bits [15:0] | Set to ((1 << *N*) - 1), where *N* is the number of output registers defined by the linked vertex shader. This count does not include generic attributes. |
| 0x0290 – 0x0293 | Write the following combinations to registers {0x0290, 0x0291, 0x0292, 0x0293}, respectively. These are used to set floating-point constants.<br>{0x00000057, 0x40800040, 0x00003F00, 0x00000000}<br>{0x00000058, 0x3D00003E, 0x000056FF, 0xFF3C0000}<br>{0x00000059, 0x3800003D, 0x00003E80, 0x003D3000}<br>{0x0000005A, 0x3CE0003B, 0x00003D80, 0x00390000}<br>{0x0000005B, 0x3C60003A, 0x80003B80, 0x00000000}<br>{0x0000005C, 0x3C98003D, 0x9C003C80, 0x003DC000}<br>{0x0000005D, 0x3DE0003E, 0x10003D80, 0x003E4000} |

When you use Loop subdivision shaders, you must also set the reserved uniforms that are assigned to the following specific registers.

**Table 8-68. Loop Subdivision Shader Uniforms and Their Registers**

| Reserved Uniform | Allocated Register |
|---|---|
| dmp_Subdivision.level | c86.x |
| dmp_Subdivision.fragmentLightingEnabled | b0 |

## 8.8.25.6. Particle System Shaders

The following table shows the register values to set when using particle system shaders.

**Table 8-69. Register Settings When Using Particle System Shaders**

| Register | Description |
|---|---|
| 0x004F, bits [2:0] | Set to 0x3. |

| | |
|---|---|
| 0x0050 – 0x0056 | Set register 0x0050 to 0x03020100. Set register 0x0051 to 0x0B0A0908. Set register 0x0052 to 0x17160D0C when texture coordinate 2 is used or 0x1F1F0D0C otherwise. Set registers 0x0053 through 0x0056 to 0x1F1F1F1F. |
| 0x0064 | Set to 0x00000000. |
| 0x006F | Set to 0x00000503 when texture coordinate 2 is used or 0x00000103 otherwise. |
| 0x0229, bits [31:31] | Set to 0. |
| 0x0242, bits [3:0] | Set to one less than the number of vertex attributes to input to the linked vertex shader. |
| 0x024A, bits [3:0] | Set to 0x4. |
| 0x0251, bits [3:0] | Set to 0x4. |
| 0x0252 | Set to 0x01004302. |
| 0x0254, bits [4:0] | No setting required. |
| 0x025E, bits [3:0] | Set to 0x2. |
| 0x0280, bits [15:0] | Set to 0x0000. |
| 0x0281, bits [23:0] | No setting required. |
| 0x0282, bits [23:0] | No setting required. |
| 0x0283, bits [23:0] | No setting required. |
| 0x0284, bits [23:0] | Set to 0x0100FE. |
| 0x0289, bits [3:0] | Set to 0x4 because the vertex shader outputs a total of five vertex attributes: the vertex coordinates and the four bounding-box sizes for the control points. |
| 0x0289, bits [15:8] | Set to 0x01. |
| 0x0289, bits [31:24] | Set to 0x08. |
| 0x028D, bits [15:0] | Set to 0x0007. |
| 0x0290 – 0x0293 | Write the following combinations to registers {0x0290, 0x0291, 0x0292, 0x0293}, respectively. These are used to set floating-point constants.<br>{0x0000004C, 0x3F0000BF, 0x00003F00, 0x00000000}<br>{0x0000004D, 0x40921F3C, 0x45F34192, 0x1F3E0000}<br>{0x0000005D, 0x3F00003F, 0x0000BC55, 0x55BE0000}<br>{0x0000005E, 0x3811113A, 0x5555B2A0, 0x1AB56C16}<br>{0x0000005F, 0x2C71DE2F, 0xA01AA5AE, 0x64A927E4} |

When you use particle system shaders, you must also set the reserved uniforms that are assigned to the following specific registers.

**Table 8-70. Particle System Shader Uniforms and Their Registers**

| Reserved Uniform | Allocated Register |
|---|---|
| dmp_PartSys.color | c26.xyzw – c29.xyzw |
| dmp_PartSys.viewport | c30.xy |
| dmp_PartSys.pointSize | c31.xy |
| dmp_PartSys.time | c31.z |
| dmp_PartSys.speed | c31.w |
| dmp_PartSys.distanceAttenuation | c32.xyz |
| dmp_PartSys.countMax | c32.w |
| dmp_PartSys.randSeed | c33.xyzw |
| dmp_PartSys.aspect | c34.xyzw – c37.xyzw |
| dmp_PartSys.randomCore | c38.xyzw |

# 8.9. Format Conversions for Values Set to Registers

Some of the values set to registers undergo internal format conversion from the values set by the application. Most of these involve conversion from 32-bit floating-point numbers.

## 8.9.1. Converting to a 24-Bit Floating-Point Number

The following code converts a 32-bit floating-point number into a 24-bit floating-point number (with a 1-bit sign, 7-bit exponent, and 16-bit significand). When you pass a 32-bit floating-point number into `_inarg`, a 24-bit floating-point number is stored as an `unsigned int` value in `_outarg`.

**Code 8-23. Converting to a 24-Bit Floating-Point Number**

```
#define UTL_F2F_16M7E(_inarg, _outarg) \
{ \
    unsigned uval_, m_; \
    int e_; \
    float f_; \
    static const int bias_ = 128 - (1 << (7 - 1)); \
    f_ = (_inarg); \
    uval_ = *(unsigned*)&f_; \
    e_ = (uval_ & 0x7fffffff) ? (((uval_ >> 23) & 0xff) - bias_) : 0; \
    m_ = (uval_ & 0x7fffff) >> (23 - 16); \
    if (e_ >= 0) \
        _outarg = m_ | (e_ << 16) | ((uval_ >> 31) << (16 + 7)); \
    else \
        _outarg = ((uval_ >> 31) << (16 + 7)); \
}
```

## 8.9.2. Converting to a 16-Bit Floating-Point Number

The following code converts a 32-bit floating-point number into a 16-bit floating-point number (with a 1-bit sign, 5-bit exponent, and 10-bit significand). When you pass a 32-bit floating-point number into `_inarg`, a 16-bit floating-point number is stored as an `unsigned int` value in `_outarg`.

**Code 8-24. Converting to a 16-Bit Floating-Point Number**

```
#define UTL_F2F_10M5E(_inarg, _outarg) \
{ \
    unsigned uval_, m_; \
    int e_; \
    float f_; \
    static const int bias_ = 128 - (1 << (5 - 1)); \
    f_ = (_inarg); \
    uval_ = *(unsigned*)&f_; \
    e_ = (uval_ & 0x7fffffff) ? (((uval_ >> 23) & 0xff) - bias_) : 0; \
    m_ = (uval_ & 0x7fffff) >> (23 - 10); \
    if (e_ >= 0) \
        _outarg = m_ | (e_ << 10) | ((uval_ >> 31) << (10 + 5)); \
    else \
        _outarg = ((uval_ >> 31) << (10 + 5)); \
}
```

## 8.9.3. Converting to a 31-Bit Floating-Point Number

The following code converts a 32-bit floating-point number into a 31-bit floating-point number (with a 1-bit sign, 7-bit exponent, and 23-bit significand). When you pass a 32-bit floating-point number into `_inarg`, a 31-bit floating-point number is stored as an `unsigned int` value in `_outarg`.

**Code 8-25. Converting to a 31-Bit Floating-Point Number**

```
#define UTL_F2F_23M7E(_inarg, _outarg) \
{ \
    unsigned uval_, m_; \
    int e_; \
    float f_; \
    static const int bias_ = 128 - (1 << (7 - 1)); \
    f_ = (_inarg); \
    uval_ = *(unsigned*)&f_; \
    e_ = (uval_ & 0x7fffffff) ? (((uval_ >> 23) & 0xff) - bias_) : 0; \
    m_ = (uval_ & 0x7fffff) >> (23 - 23); \
    if (e_ >= 0) \
        _outarg = m_ | (e_ << 23) | ((uval_ >> 31) << (23 + 7)); \
    else \
        _outarg = ((uval_ >> 31) << (23 + 7)); \
}
```

## 8.9.4. Converting to a 20-Bit Floating-Point Number

The following code converts a 32-bit floating-point number into a 20-bit floating-point number (with a 1-bit sign, 7-bit exponent, and 12-bit significand). When you pass a 32-bit floating-point number into `_inarg`, a 20-bit floating-point number is stored as an `unsigned int` value in `_outarg`.

**Code 8-26. Converting to a 20-Bit Floating-Point Number**

```
#define UTL_F2F_12M_7E(_inarg, _outarg) \
{ \
    unsigned uval_, m_; \
    int e_; \
```

```
    float f_; \
    static const int bias_ = 128 - (1 << (7 - 1)); \
    f_ = (_inarg); \
    uval_ = *(unsigned*)&f_; \
    e_ = (uval_ & 0x7fffffff) ? (((uval_ >> 23) & 0xff) - bias_) : 0; \
    m_ = (uval_ & 0x7fffff) >> (23 - 12); \
    if (e_ >= 0) \
        _outarg = m_ | (e_ << 12) | ((uval_ >> 31) << (12 + 7)); \
    else \
        _outarg = ((uval_ >> 31) << (12 + 7)); \
}
```

## 8.9.5. Converting to an 8-Bit Signed Fixed-Point Number With 7 Fractional Bits

The following code converts a 32-bit floating-point number into a 8-bit signed fixed-point number with 7 fractional bits. Negative values are represented in two's complement. If you pass a 32-bit floating-point number to `_inarg`, a 8-bit fixed-point number is stored in `_outarg`.

**Code 8-27. Converting to a 8-Bit Signed Fixed-Point Number With 7 Fractional Bits**

```
#define UTL_F2FX_8W_1I_T(_inarg, _outarg) \
{ \
    float f_; \
    unsigned v_; \
    f_ = (_inarg); \
    v_ = *(unsigned*)&f_; \
    if (f_ == 0.f || (v_ & 0x7f800000) == 0x7f800000) \
        _outarg = 0; \
    else \
    { \
        f_ += 0.5f * (1 << 1); \
        f_ *= 1 << (8 - 1); \
        if (f_ < 0) \
            f_ = 0; \
        else if (f_ >= (1 << 8)) \
            f_ = (1 << 8) - 1; \
        if (f_ >= (1 << (8 - 1))) \
            _outarg = (unsigned)(f_ - (1 << (8 - 1))); \
        else \
            _outarg = (unsigned)(f_ + (1 << (8 - 1))); \
    } \
}
```

## 8.9.6. Converting to an 12-Bit Signed Fixed-Point Number With 11 Fractional Bits

The following code converts a 32-bit floating-point number into a 12-bit signed fixed-point number with 11 fractional bits. Because the fractional bits are absolute values, negative values are not represented in two's complement. If you pass a 32-bit floating-point number to `_inarg`, a 12-bit fixed-point number is stored in `_outarg`.

**Code 8-28. Converting to a 12-Bit Signed Fixed-Point Number With 11 Fractional Bits**

```
#define UTL_F2FX_12W_1I_F(_inarg, _outarg) \
{ \
    float f_; \
    unsigned v_; \
```

```
    f_ = (_inarg); \
    v_ = *(unsigned*)&f_; \
    if (f_ == 0.f || (v_ & 0x7f800000) == 0x7f800000) \
        _outarg = 0; \
    else \
    { \
        f_ *= (1 << (12 - 1)); \
        if (f_ < 0) \
        { \
            _outarg = 1 << (12 - 1); \
            f_ = -f_; \
        } \
        else \
            _outarg = 0; \
        if (f_ >= (1 << (12 - 1))) f_ = (1 << (12 - 1)) - 1; \
            _outarg |= (unsigned)(f_); \
    } \
}
```

## 8.9.7. Converting to a 12-Bit Signed Fixed-Point Number With 11 Fractional Bits

The following code converts a 32-bit floating-point number into a 12-bit signed fixed-point number with 11 fractional bits. Negative values are represented in two's complement. If you pass a 32-bit floating-point number to `_inarg`, a 12-bit fixed-point number is stored in `_outarg`.

**Code 8-29. Converting to a 12-Bit Signed Fixed-Point Number With 11 Fractional Bits**

```
#define UTL_F2FX_12W_1I_T(_inarg, _outarg) \
{ \
    float f_; \
    unsigned v_; \
    f_ = (_inarg); \
    v_ = *(unsigned*)&f_; \
    if (f_ == 0.f || (v_ & 0x7f800000) == 0x7f800000) \
        _outarg = 0; \
    else \
    { \
        f_ += 0.5f * (1 << 1); \
        f_ *= 1 << (12 - 1); \
        if (f_ < 0) \
            f_ = 0; \
        else if (f_ >= (1 << 12)) \
            f_ = (1 << 12) - 1; \
        if (f_ >= (1 << (12 - 1))) \
            _outarg = (unsigned)(f_ - (1 << (12 - 1))); \
        else \
            _outarg = (unsigned)(f_ + (1 << (12 - 1))); \
    } \
}
```

## 8.9.8. Converting to an 13-Bit Signed Fixed-Point Number With 8 Fractional Bits

The following code converts a 32-bit floating-point number into a 13-bit signed fixed-point number with 8 fractional bits. Negative values are represented in two's complement. If you pass a 32-bit floating-point number to `_inarg`, a 13-bit fixed-point number is stored in `_outarg`.

**Code 8-30. Converting to a 13-Bit Signed Fixed-Point Number With 8 Fractional Bits**

```
#define UTL_F2FX_13W_5I_T(_inarg, _outarg) \
{ \
    float f_; \
    unsigned v_; \
    f_ = (_inarg); \
    v_ = *(unsigned*)&f_; \
    if (f_ == 0.f || (v_ & 0x7f800000) == 0x7f800000) \
        _outarg = 0; \
    else \
    { \
        f_ += 0.5f * (1 << 5); \
        f_ *= 1 << (13 - 5); \
        if (f_ < 0) \
            f_ = 0; \
        else if (f_ >= (1 << 13)) \
            f_ = (1 << 13) - 1; \
        if (f_ >= (1 << (13 - 1))) \
            _outarg = (unsigned)(f_ - (1 << (13 - 1))); \
        else \
            _outarg = (unsigned)(f_ + (1 << (13 - 1))); \
    } \
}
```

## 8.9.9. Converting to an 13-Bit Signed Fixed-Point Number With 11 Fractional Bits

The following code converts a 32-bit floating-point number into a 13-bit signed fixed-point number with 11 fractional bits. Negative values are represented in two's complement. If you pass a 32-bit floating-point number to `_inarg`, a 13-bit fixed-point number is stored in `_outarg`.

**Code 8-31. Converting to a 13-Bit Signed Fixed-Point Number With 11 Fractional Bits**

```
#define UTL_F2FX_13W_2I_T(_inarg, _outarg) \
{ \
    float f_; \
    unsigned v_; \
    f_ = (_inarg); \
    v_ = *(unsigned*)&f_; \
    if (f_ == 0.f || (v_ & 0x7f800000) == 0x7f800000) \
        _outarg = 0; \
    else \
    { \
        f_ += 0.5f * (1 << 2); \
        f_ *= 1 << (13 - 2); \
        if (f_ < 0) \
            f_ = 0; \
        else if (f_ >= (1 << 13)) \
            f_ = (1 << 13) - 1; \
        if (f_ >= (1 << (13 - 1))) \
            _outarg = (unsigned)(f_ - (1 << (13 - 1))); \
        else \
            _outarg = (unsigned)(f_ + (1 << (13 - 1))); \
    } \
}
```

## 8.9.10. Converting to an 16-Bit Signed Fixed-Point Number With 12 Fractional Bits

The following code converts a 32-bit floating-point number into a 16-bit signed fixed-point number with 12 fractional bits. Negative values are represented in two's complement. If you pass a 32-bit

floating-point number to `_inarg`, a 16-bit fixed-point number is stored in `_outarg`.

**Code 8-32. Converting to a 16-Bit Signed Fixed-Point Number With 12 Fractional Bits**

```
#define UTL_F2FX_16W_4I_T(_inarg, _outarg) \
{ \
    float f_; \
    unsigned v_; \
    f_ = (_inarg); \
    v_ = *(unsigned*)&f_; \
    if (f_ == 0.f || (v_ & 0x7f800000) == 0x7f800000) \
        _outarg = 0; \
    else \
    { \
        f_ += 0.5f * (1 << 4); \
        f_ *= 1 << (16 - 4); \
        if (f_ < 0) \
            f_ = 0; \
        else if (f_ >= (1 << 16)) \
            f_ = (1 << 16) - 1; \
        if (f_ >= (1 << (16 - 1))) \
            _outarg = (unsigned)(f_ - (1 << (16 - 1))); \
        else \
            _outarg = (unsigned)(f_ + (1 << (16 - 1))); \
    } \
}
```

## 8.9.11. Converting to a 0-Bit Signed Fixed-Point Number With 8 Fractional Bits

The following code converts a 32-bit floating-point number into a 8-bit unsigned fixed-point number with 0 fractional bits. If you pass a 32-bit floating-point number to `_inarg`, a 8-bit fixed-point number is stored in `_outarg`.

**Code 8-33. Converting to an 0-Bit Unsigned Fixed-Point Number With 8 Fractional Bits**

```
#define UTL_F2UFX_8W_8I(_inarg, _outarg) \
{ \
    float f_ = (_inarg); \
    unsigned val_; \
    unsigned v_ = *(unsigned*)&f_; \
    if (f_ <= 0 || (v_ & 0x7f800000) == 0x7f800000) \
        val_ = 0; \
    else \
    { \
        f_ *= 1 << (8 - 8); \
        if (f_ >= (1 << 8)) \
            val_ = (1 << 8) - 1; \
        else \
            val_ = (unsigned)(f_); \
    } \
    (_outarg) = val_; \
}
```

## 8.9.12. Converting to an 11-Bit Signed Fixed-Point Number With 11 Fractional Bits

The following code converts a 32-bit floating-point number into a 11-bit unsigned fixed-point

number with 11 fractional bits. If you pass a 32-bit floating-point number to `_inarg`, a 11-bit fixed-point number is stored in `_outarg`.

**Code 8-34. Converting to an 11-Bit Unsigned Fixed-Point Number With 11 Fractional Bits**

```
#define UTL_F2UFX_11W_0I(_inarg, _outarg) \
{ \
    float f_ = (_inarg); \
    unsigned val_; \
    unsigned v_ = *(unsigned*)&f_; \
    if (f_ <= 0 || (v_ & 0x7f800000) == 0x7f800000) \
        val_ = 0; \
    else \
    { \
        f_ *= 1 << (11 - 0); \
        if (f_ >= (1 << 11)) \
            val_ = (1 << 11) - 1; \
        else \
            val_ = (unsigned)(f_); \
    } \
    (_outarg) = val_; \
}
```

## 8.9.13. Converting to an 12-Bit Signed Fixed-Point Number With 12 Fractional Bits

The following code converts a 32-bit floating-point number into a 12-bit unsigned fixed-point number with 12 fractional bits. If you pass a 32-bit floating-point number to `_inarg`, a 12-bit fixed-point number is stored in `_outarg`.

**Code 8-35. Converting to an 12-Bit Unsigned Fixed-Point Number With 12 Fractional Bits**

```
#define UTL_F2UFX_12W_0I(_inarg, _outarg) \
{ \
    float f_ = (_inarg); \
    unsigned val_; \
    unsigned v_ = *(unsigned*)&f_; \
    if (f_ <= 0 || (v_ & 0x7f800000) == 0x7f800000) \
        val_ = 0; \
    else \
    { \
        f_ *= 1 << (12 - 0); \
        if (f_ >= (1 << 12)) \
            val_ = (1 << 12) - 1; \
        else \
            val_ = (unsigned)(f_); \
    } \
    (_outarg) = val_; \
}
```

## 8.9.14. Converting to an 24-Bit Signed Fixed-Point Number With 24 Fractional Bits

The following code converts a 32-bit floating-point number into a 24-bit unsigned fixed-point number with 24 fractional bits. If you pass a 32-bit floating-point number to `_inarg`, a 24-bit fixed-point number is stored in `_outarg`.

**Code 8-36. Converting to a 24-Bit Unsigned Fixed-Point Number With 24 Fractional Bits**

```
#define UTL_F2UFX_24W_0I(_inarg, _outarg) \
{ \
    float f_ = (_inarg); \
    unsigned val_; \
    unsigned v_ = *(unsigned*)&f_; \
    if (f_ <= 0 || (v_ & 0x7f800000) == 0x7f800000) \
        val_ = 0; \
    else \
    { \
        f_ *= 1 << (24 - 0); \
        if (f_ >= (1 << 24)) \
            val_ = (1 << 24) - 1; \
        else \
            val_ = (unsigned)(f_); \
    } \
    (_outarg) = val_; \
}
```

## 8.9.15. Converting to an 8-Bit Signed Fixed-Point Number With 24 Fractional Bits

The following code converts a 32-bit floating-point number into a 24-bit unsigned fixed-point number with 8 fractional bits. If you pass a 32-bit floating-point number to `_inarg`, a 24-bit fixed-point number is stored in `_outarg`.

**Code 8-37. Converting to an 8-Bit Unsigned Fixed-Point Number With 24 Fractional Bits**

```
#define UTL_F2UFX_24W_16I(_inarg, _outarg) \
{ \
    float f_ = (_inarg); \
    unsigned val_; \
    unsigned v_ = *(unsigned*)&f_; \
    if (f_ <= 0 || (v_ & 0x7f800000) == 0x7f800000) \
        val_ = 0; \
    else \
    { \
        f_ *= 1 << (24 - 16); \
        if (f_ >= (1 << 24)) \
            val_ = (1 << 24) - 1; \
        else \
            val_ = (unsigned)(f_); \
    } \
    (_outarg) = val_; \
}
```

## 8.9.16. Converting From a Floating-Point Number (Between 0 and 1) to an 8-Bit Unsigned Integer

The following code converts a 32-bit floating-point number in the range from 0 through 1 into an 8-bit unsigned integer. If you pass a 32-bit floating-point number into `f`, an 8-bit unsigned integer is returned.

**Code 8-38. Converting From a Floating-Point Number (Between 0 and 1) to an 8-Bit Unsigned Integer**

```
((unsigned)(0.5f + (f) * (float)((1 << 8) - 1)))
```

## 8.9.17. Converting From a Floating-Point Number (Between 0 and 1) to an 8-Bit Unsigned Integer

The following code converts a 32-bit floating-point number in the range from 0 through 1 into an 8-bit unsigned integer. If you pass a 32-bit floating-point number into `f`, an 8-bit unsigned integer is returned.

**Code 8-39. Converting From a Floating-Point Number (Between 0 and 1) to an 8-Bit Unsigned Integer**

```
((unsigned)((f) * (float)((1 << 8) - 1)))
```

## 8.9.18. Converting From a Floating-Point Number (From –1 Through 1) to an 8-Bit Unsigned Integer

The following code converts a 32-bit floating-point number in the range from –1 through 1 into an 8-bit signed integer. If you pass a 32-bit floating-point number into `f`, an 8-bit signed integer is returned.

**Code 8-40. Converting From a Floating-Point Number (From –1 through 1) to an 8-Bit Signed Integer**

```
(((unsigned int)(fabs(127.f * (f))) & 0x7f)|(f < 0 ? 0x80 : 0))
```

## 8.9.19. Converting From a 16-Bit Floating-Point Number to a 32-Bit Floating-Point Number

The following code converts a 16-bit floating-point number (with one sign bit, a 5-bit exponent, and a 10-bit significand) into a 32-bit floating-point number. If you pass a 16-bit floating-point number stored as an `unsigned int` to `_inarg`, a 32-bit floating-point number is stored in the `float` variable specified by `_outarg`.

**Code 8-41. Converting From a 16-Bit Floating-Point Number to a 32-Bit Floating-Point Number**

```
#define UTL_U2F_10M5E(_inarg, _outarg) \
{ \
    int e_; \
    unsigned m_; \
    unsigned u_ = (_inarg); \
    const int width_ = 10 + 5 + 1; \
    const int bias_ = 128 - (1 << (5 - 1)); \
    e_ = (u_ >> 10) & ((1 << 5) - 1); \
    m_ = u_ & ((1 << 10) - 1); \
    if (u_ & ((1 << (width_ - 1)) - 1)) \
        u_ = ((u_ >> (5 + 10)) << 31) | (m_ << (23 - 10)) | ((e_ + bias_) <<
23); \
    else \
        u_ = ((u_ >> (5 + 10)) << 31); \
```

```
    (_outarg) = *(float*)&u_; \
}
```

# 8.10. Register Map

This section lists register maps for the registers described in 8.8. PICA Register Information. The register maps are organized by register address, and they include the related section to see in this document, related functions or uniforms, and the states in which the register may be updated.

## 8.10.1. Register Map for Registers 0x0010 – 0x00FF

**Table 8-71. Register Map (Addresses 0x0010 – 0x00FF)**

| Address | Section | Function / Reserved Uniform | NN_GX_STA |
|---|---|---|---|
| 0x0010 | 8.8.22 | nngxSplitDrawCmdlist, nngxTransferRenderImage | - |
| 0x0040 | 8.8.15 | glCullFace, glDisable(GL_CULL_FACE), glEnable(GL_CULL_FACE), glFrontFace | OTHERS |
| 0x0041 through 0x0044 | 8.8.10 | `glViewport` | OTHERS |
| 0x0047 | 8.8.9.4 | `dmp_FragOperation.enableClippingPlane` | FSUNIFORM |
| 0x0048 through 0x004B | | `dmp_FragOperation.clippingPlane` | FSUNIFORM |
| 0x004D | 8.8.9.3 | dmp_FragOperation.wScale, glDepthRangef, glDisable(GL_POLYGON_OFFSET_FILL), glEnable(GL_POLYGON_OFFSET_FILL), glPolygonOffset | FSUNIFORM TRIOFFSET |
| 0x004E | | | |
| 0x004F | 8.8.1.10 8.8.20.8 | Sets the number of used output registers. | SHADERPRO |
| 0x0050 through 0x0056 | 8.8.1.12 8.8.20.10 | Output register attributes settings. | SHADERPRO |
| 0x0061 | 8.8.13 | `glEarlyDepthFuncDMP` | OTHERS |
| 0x0062 | | glDisable(GL_EARLY_DEPTH_TEST_DMP), glEnable(GL_EARLY_DEPTH_TEST_DMP) | OTHERS |
| 0x0063 | | `glClear(GL_EARLY_DEPTH_BUFFER_BIT_DMP)` | - |
| 0x0064 | 8.8.1.12 8.8.20.10 | Output register attributes settings. | SHADERPRO |
| 0x0065 through 0x0067 | 8.8.16 | glDisable(GL_SCISSOR_TEST), glEnable(GL_SCISSOR_TEST), glScissor | SCISSOR |
| 0x0068 | 8.8.10 | `glViewport` | OTHERS |
| 0x006A | 8.8.13 | `glClearEarlyDepthDMP` | OTHERS |
| 0x006D | 8.8.9.3 | `dmp_FragOperation.wScale` | FSUNIFORM |
| 0x006E | 8.8.3 | glRenderbufferStorage, glTexture2DImage2D | FRAMEBUFFI |

| | | | |
|---|---|---|---|
| 0x006F | 8.8.1.13<br>8.8.20.11 | Output attributes clock control. | SHADERPRO |
| 0x0080 | 8.8.6.2 | dmp_Texture[i].samplerType (i=0,1,2) | TEXTURE |
| | 8.8.6.3 | dmp_Texture[2].texcoord, dmp_Texture[3].texcoord, dmp_Texture[3].samplerType | FSUNIFORM |
| | 8.8.19 | glDrawArrays, glDrawElements | - |
| 0x0081 | 8.8.6.8 | glTexParameter | TEXTURE |
| 0x0082 | 8.8.6.6 | glTexImage2D, glCompressedTexImage2D, glCopyTexImage2D | TEXTURE |
| 0x0083 | 8.8.6.2 | dmp_Texture[0].samplerType | TEXTURE |
| | 8.8.6.7 | glTexImage2D, glCompressedTexImage2D, glCopyTexImage2D | TEXTURE |
| | 8.8.6.8 | glTexParameter | TEXTURE |
| 0x0084 | 8.8.6.8 | glTexParameter, glCopyTexImage2D, glCompressedTexImage2D, glTexImage2D | TEXTURE |
| 0x0085 through 0x008A | 8.8.2 | glTexImage2D, glCompressedTexImage2D, glCopyTexImage2D | TEXTURE |
| 0x008B | 8.8.6.1 | dmp_Texture[0].perspectiveShadow, dmp_Texture[0].shadowZBias | FSUNIFORM |
| 0x008E | 8.8.6.7 | glTexImage2D, glCompressedTexImage2D, glCopyTexImage2D | TEXTURE |
| 0x008F | 8.8.5.1 | dmp_FragmentLighting.enabled | FSUNIFORM |
| 0x0091 | 8.8.6.8 | glTexParameter | TEXTURE |
| 0x0092 | 8.8.6.6 | glTexImage2D, glCompressedTexImage2D, glCopyTexImage2D | TEXTURE |
| 0x0093 | 8.8.6.7 | glTexImage2D, glCompressedTexImage2D, glCopyTexImage2D | TEXTURE |
| | 8.8.6.8 | glTexParameter | TEXTURE |
| 0x0094 | 8.8.6.8 | glTexImage2D, glCompressedTexImage2D, glCopyTexImage2D | TEXTURE |
| 0x0095 | 8.8.2 | glTexImage2D, glCompressedTexImage2D, glCopyTexImage2D | TEXTURE |
| 0x0096 | 8.8.6.7 | glTexImage2D, glCompressedTexImage2D, glCopyTexImage2D | TEXTURE |
| 0x0099 | 8.8.6.8 | glTexParameter | TEXTURE |
| 0x009A | 8.8.6.6 | glTexImage2D, glCompressedTexImage2D, glCopyTexImage2D | TEXTURE |
| 0x009B | 8.8.6.7 | glTexImage2D, glCompressedTexImage2D, glCopyTexImage2D | TEXTURE |
| | 8.8.6.8 | glTexParameter | TEXTURE |
| 0x009C | 8.8.6.8 | glTexParameter, glCopyTexImage2D, glCompressedTexImage2D, glTexImage2D | TEXTURE |
| 0x009D | 8.8.2 | glTexImage2D, glCompressedTexImage2D, glCopyTexImage2D | TEXTURE |
| 0x009E | 8.8.6.7 | glTexImage2D, glCompressedTexImage2D, glCopyTexImage2D | TEXTURE |
| 0x00A8 | 8.8.6.4 | dmp_Texture[3].ptClampU, dmp_Texture[3].ptClampV, dmp_Texture[3].ptRgbMap, dmp_Texture[3].ptAlphaMap, dmp_Texture[3].ptAlphaSeparate, dmp_Texture[3].ptNoiseEnable, dmp_Texture[3].ptShiftU, dmp_Texture[3].ptShiftV, dmp_Texture[3].ptTexBias | FSUNIFORM |
| 0x00A9 through 0x00AB | | dmp_Texture[3].ptNoiseU, dmp_Texture[3].ptNoiseV | FSUNIFORM |
| 0x00AC | | dmp_Texture[3].ptMinFilter, dmp_Texture[3].ptTexWidth, dmp_Texture[3].ptTexBias | FSUNIFORM |
| 0x00AD | | dmp_Texture[3].ptTexOffset | FSUNIFORM |
| 0x00AF | 8.8.6.5 | dmp_Texture[3].ptSampler{RgbMap,AlphaMap,NoiseMap,R,G,B,A} | LUT |
| 0x00B0 through 0x00B7 | | Sets data to lookup tables. | LUT |

| | | | |
|---|---|---|---|
| 0x00C0 | 8.8.4 | dmp_TexEnv[0].srcRgb, dmp_TexEnv[0].srcAlpha | FSUNIFORM |
| 0x00C1 | | dmp_TexEnv[0].operandRgb, dmp_TexEnv[0].operandAlpha | FSUNIFORM |
| 0x00C2 | | dmp_TexEnv[0].combineRgb, dmp_TexEnv[0].combineAlpha | FSUNIFORM |
| 0x00C3 | | dmp_TexEnv[0].constRgba | FSUNIFORM |
| 0x00C4 | | dmp_TexEnv[0].scaleRgb, dmp_TexEnv[0].scaleAlpha | FSUNIFORM |
| 0x00C8 | 8.8.4 | dmp_TexEnv[1].srcRgb, dmp_TexEnv[1].srcAlpha | FSUNIFORM |
| 0x00C9 | | dmp_TexEnv[1].operandRgb, dmp_TexEnv[1].operandAlpha | FSUNIFORM |
| 0x00CA | | dmp_TexEnv[1].combineRgb, dmp_TexEnv[1].combineAlpha | FSUNIFORM |
| 0x00CB | | dmp_TexEnv[1].constRgba | FSUNIFORM |
| 0x00CC | | dmp_TexEnv[1].scaleRgb, dmp_TexEnv[1].scaleAlpha | FSUNIFORM |
| 0x00D0 | 8.8.4 | dmp_TexEnv[2].srcRgb, dmp_TexEnv[2].srcAlpha | FSUNIFORM |
| 0x00D1 | | dmp_TexEnv[2].operandRgb, dmp_TexEnv[2].operandAlpha | FSUNIFORM |
| 0x00D2 | | dmp_TexEnv[1].combineRgb, dmp_TexEnv[1].combineAlpha | FSUNIFORM |
| 0x00D3 | | dmp_TexEnv[2].constRgba | FSUNIFORM |
| 0x00D4 | | dmp_TexEnv[2].scaleRgb, dmp_TexEnv[2].scaleAlpha | FSUNIFORM |
| 0x00D8 | 8.8.4 | dmp_TexEnv[3].srcRgb, dmp_TexEnv[3].srcAlpha | FSUNIFORM |
| 0x00D9 | | dmp_TexEnv[3].operandRgb, dmp_TexEnv[3].operandAlpha | FSUNIFORM |
| 0x00DA | | dmp_TexEnv[3].combineRgb, dmp_TexEnv[3].combineAlpha | FSUNIFORM |
| 0x00DB | | dmp_TexEnv[3].constRgba | FSUNIFORM |
| 0x00DC | | dmp_TexEnv[3].scaleRgb, dmp_TexEnv[3].scaleAlpha | FSUNIFORM |
| 0x00E0 | 8.8.4.1 | dmp_TexEnv[i].bufferInput (i=1,2,3,4) | FSUNIFORM |
| | 8.8.7.1 | dmp_Gas.shadingDensitySrc | FSUNIFORM |
| | 8.8.8.1 | dmp_Fog.mode, dmp_Fog.zFlip | FSUNIFORM |
| 0x00E1 | 8.8.8.1 | dmp_Fog.color | FSUNIFORM |
| 0x00E4 | 8.8.7.1 | dmp_Gas.attenuation | FSUNIFORM |
| 0x00E5 | | dmp_Gas.accMax | FSUNIFORM |
| 0x00E6 | 8.8.8.2 | dmp_Fog.sampler | LUT |
| 0x00E8 through 0x00EF | | Sets data to lookup tables. | LUT |
| 0x00F0 | 8.8.4 | dmp_TexEnv[4].srcRgb, dmp_TexEnv[4].srcAlpha | FSUNIFORM |
| 0x00F1 | | dmp_TexEnv[4].operandRgb, dmp_TexEnv[4].operandAlpha | FSUNIFORM |
| 0x00F2 | | dmp_TexEnv[4].combineRgb, dmp_TexEnv[4].combineAlpha | FSUNIFORM |
| 0x00F3 | | dmp_TexEnv[4].constRgba | FSUNIFORM |
| 0x00F4 | | dmp_TexEnv[4].scaleRgb, dmp_TexEnv[4].scaleAlpha | FSUNIFORM |
| 0x00F8 | 8.8.4 | dmp_TexEnv[5].srcRgb, dmp_TexEnv[5].srcAlpha | FSUNIFORM |
| 0x00F9 | | dmp_TexEnv[5].operandRgb, dmp_TexEnv[5].operandAlpha | FSUNIFORM |
| 0x00FA | | dmp_TexEnv[5].combineRgb, dmp_TexEnv[5].combineAlpha | FSUNIFORM |
| 0x00FB | | dmp_TexEnv[5].constRgba | FSUNIFORM |
| 0x00FC | | dmp_TexEnv[5].scaleRgb, dmp_TexEnv[5].scaleAlpha | FSUNIFORM |
| 0x00FD | 8.8.4.1 | dmp_TexEnv[0].bufferColor | FSUNIFORM |

## 8.10.2. Register Map for Registers 0x0100 – 0x01FF

**Table 8-72. Register Map (Addresses 0x0100 – 0x01FF)**

| Address | Section | Function / Reserved Uniform | NN_GX_STATE_ |
|---|---|---|---|
| 0x0100 | 8.8.9.1 | dmp_FragOperation.mode | FSUNIFORM |
| | 8.8.12 | glDisable(GL_BLEND), glEnable(GL_BLEND), glDisable(GL_COLOR_LOGIC_OP), glEnable(GL_COLOR_LOGIC_OP) | OTHERS |
| 0x0101 | 8.8.12 | glBlendEquation, glBlendEquationSeparate, glBlendFunc, glBlendFuncSeparate | OTHERS |
| 0x0102 | | glLogicOp | OTHERS |
| 0x0103 | | glBlendColor | OTHERS |
| 0x0104 | 8.8.9.5 | dmp_FragOperation.enableAlphaTest, dmp_FragOperation.alphaTestFunc, dmp_FragOperation.alphaRefValue | FSUNIFORM |
| 0x0105 | 8.8.14 | glDisable(GL_STENCIL_TEST), glEnable(GL_STENCIL_TEST), glStencilFunc, glStencilMask | OTHERS |
| 0x0106 | | glStencilOp | OTHERS |
| 0x0107 | 8.8.11 | glDisable(GL_DEPTH_TEST), glEnable(GL_DEPTH_TEST), glDepthFunc, glDepthMask | OTHERS |
| | 8.8.17 | glColorMask | OTHERS |
| 0x0110 | 8.8.21 | glFinish, glFlush, nngxSplitDrawCmdlist, nngxTransferRenderImage | FRAMEBUFFER FBACCESS |
| 0x0111 | | glFinish, glFlush, glDrawArrays, glDrawElements, nngxSplitDrawCmdlist, nngxTransferRenderImage | FRAMEBUFFER FBACCESS |
| 0x0112 through 0x0115 | 8.8.9.6 | dmp_FragOperation.mode, glDisable(GL_BLEND), glEnable(GL_BLEND), glDisable(GL_COLOR_LOGIC_OP), glEnable(GL_COLOR_LOGIC_OP), glColorMask, glDisable(GL_DEPTH_TEST), glEnable(GL_DEPTH_TEST), glDepthMask, glDisable(GL_STENCIL_TEST), glEnable(GL_STENCIL_TEST), glStencilMask | FBACCESS |
| 0x0116 | 8.8.3 | glRenderbufferStorage, glTexture2DImage2D | FRAMEBUFFER |
| 0x0117 | | | |
| 0x0118 | 8.8.13 | glDisable(GL_EARLY_DEPTH_TEST_DMP), glEnable(GL_EARLY_DEPTH_TEST_DMP) | OTHERS |
| 0x011B | 8.8.18 | glRenderBlockModeDMP | OTHERS |
| 0x011C through 0x011E | 8.8.3 | glRenderbufferStorage, glTexImage2D, glTexture2DImage2D | FRAMEBUFFER |
| 0x0120 | 8.8.7.1 | dmp_Gas.lightXY | FSUNIFORM |
| 0x0121 | | dmp_Gas.lightZ | FSUNIFORM |
| 0x0122 | | | |
| 0x0123 | 8.8.7.2 | dmp_Gas.sampler{TR,TG,TB} | LUT |
| 0x0124 | | Sets data to lookup tables. | LUT |
| 0x0125 | 8.8.7.1 | dmp_Gas.autoAcc | - |
| 0x0126 | 8.8.7.1 | dmp_Gas.deltaZ | FSUNIFORM |
| | 8.8.11 | glDepthFunc | OTHERS |
| 0x0130 | 8.8.9.2 | dmp_FragOperation.penumbraScale, | FSUNIFORM |

| | | | |
|---|---|---|---|
| | | dmp_FragOperation.penumbraBias | |
| 0x0140 | 8.8.5.3 | dmp_FragmentMaterial.specular0,<br>dmp_FragmentLightSource[0].specular0 | FSUNIFORM |
| 0x0141 | | dmp_LightEnv.lutEnabledRefl, dmp_FragmentMaterial.specular1,<br>dmp_FragmentLightSource[0].specular1 | FSUNIFORM |
| 0x0142 | | dmp_FragmentMaterial.diffuse,<br>dmp_FragmentLightSource[0].diffuse | FSUNIFORM |
| 0x0143 | | dmp_FragmentMaterial.ambient,<br>dmp_FragmentLightSource[0].ambient | FSUNIFORM |
| 0x0144 | | dmp_FragmentLightSource[0].position | FSUNIFORM |
| 0x0145 | | | |
| 0x0146 | | dmp_FragmentLightSource[0].spotDirection | FSUNIFORM |
| 0x0147 | | | |
| 0x0149 | | dmp_FragmentLightSource[0].position,<br>dmp_FragmentLightSource[0].twoSideDiffuse,<br>dmp_FragmentLightSource[0].geomFactor0,<br>dmp_FragmentLightSource[0].geomFactor1 | FSUNIFORM |
| 0x014A | | dmp_FragmentLightSource[0].distanceAttenuationBias | FSUNIFORM |
| 0x014B | | dmp_FragmentLightSource[0].distanceAttenuationScale | FSUNIFORM |
| 0x0150 | 8.8.5.3 | dmp_FragmentMaterial.specular0,<br>dmp_FragmentLightSource[1].specular0 | FSUNIFORM |
| 0x0151 | | dmp_LightEnv.lutEnabledRefl, dmp_FragmentMaterial.specular1,<br>dmp_FragmentLightSource[1].specular1 | FSUNIFORM |
| 0x0152 | | dmp_FragmentMaterial.diffuse,<br>dmp_FragmentLightSource[1].diffuse | FSUNIFORM |
| 0x0153 | | dmp_FragmentMaterial.ambient,<br>dmp_FragmentLightSource[1].ambient | FSUNIFORM |
| 0x0154 | | dmp_FragmentLightSource[1].position | FSUNIFORM |
| 0x0155 | | | |
| 0x0156 | | dmp_FragmentLightSource[1].spotDirection | FSUNIFORM |
| 0x0157 | | | |
| 0x0159 | | dmp_FragmentLightSource[1].position,<br>dmp_FragmentLightSource[1].twoSideDiffuse,<br>dmp_FragmentLightSource[1].geomFactor0,<br>dmp_FragmentLightSource[1].geomFactor1 | FSUNIFORM |
| 0x015A | | dmp_FragmentLightSource[1].distanceAttenuationBias | FSUNIFORM |
| 0x015B | | dmp_FragmentLightSource[1].distanceAttenuationScale | FSUNIFORM |
| 0x0160 | 8.8.5.3 | dmp_FragmentMaterial.specular0,<br>dmp_FragmentLightSource[2].specular0 | FSUNIFORM |
| 0x0161 | | dmp_LightEnv.lutEnabledRefl, dmp_FragmentMaterial.specular1,<br>dmp_FragmentLightSource[2].specular1 | FSUNIFORM |
| 0x0162 | | dmp_FragmentMaterial.diffuse,<br>dmp_FragmentLightSource[2].diffuse | FSUNIFORM |
| 0x0163 | | dmp_FragmentMaterial.ambient,<br>dmp_FragmentLightSource[2].ambient | FSUNIFORM |
| 0x0164 | | dmp_FragmentLightSource[2].position | FSUNIFORM |
| 0x0165 | | | |
| 0x0166 | | dmp_FragmentLightSource[2].spotDirection | FSUNIFORM |
| 0x0167 | | | |
| | | dmp_FragmentLightSource[2].position, | |

| | | | |
|---|---|---|---|
| 0x0169 | | dmp_FragmentLightSource[2].twoSideDiffuse,<br>dmp_FragmentLightSource[2].geomFactor0,<br>dmp_FragmentLightSource[2].geomFactor1 | FSUNIFORM |
| 0x016A | | dmp_FragmentLightSource[2].distanceAttenuationBias | FSUNIFORM |
| 0x016B | | dmp_FragmentLightSource[2].distanceAttenuationScale | FSUNIFORM |
| 0x0170 | 8.8.5.3 | dmp_FragmentMaterial.specular0,<br>dmp_FragmentLightSource[3].specular0 | FSUNIFORM |
| 0x0171 | | dmp_LightEnv.lutEnabledRefl, dmp_FragmentMaterial.specular1,<br>dmp_FragmentLightSource[3].specular1 | FSUNIFORM |
| 0x0172 | | dmp_FragmentMaterial.diffuse,<br>dmp_FragmentLightSource[3].diffuse | FSUNIFORM |
| 0x0173 | | dmp_FragmentMaterial.ambient,<br>dmp_FragmentLightSource[3].ambient | FSUNIFORM |
| 0x0174 | | dmp_FragmentLightSource[3].position | FSUNIFORM |
| 0x0175 | | | |
| 0x0176 | | dmp_FragmentLightSource[3].spotDirection | FSUNIFORM |
| 0x0177 | | | |
| 0x0179 | | dmp_FragmentLightSource[3].position,<br>dmp_FragmentLightSource[3].twoSideDiffuse,<br>dmp_FragmentLightSource[3].geomFactor0,<br>dmp_FragmentLightSource[3].geomFactor1 | FSUNIFORM |
| 0x017A | | dmp_FragmentLightSource[3].distanceAttenuationBias | FSUNIFORM |
| 0x017B | | dmp_FragmentLightSource[3].distanceAttenuationScale | FSUNIFORM |
| 0x0180 | 8.8.5.3 | dmp_FragmentMaterial.specular0,<br>dmp_FragmentLightSource[4].specular0 | FSUNIFORM |
| 0x0181 | | dmp_LightEnv.lutEnabledRefl, dmp_FragmentMaterial.specular1,<br>dmp_FragmentLightSource[4].specular1 | FSUNIFORM |
| 0x0182 | | dmp_FragmentMaterial.diffuse,<br>dmp_FragmentLightSource[4].diffuse | FSUNIFORM |
| 0x0183 | | dmp_FragmentMaterial.ambient,<br>dmp_FragmentLightSource[4].ambient | FSUNIFORM |
| 0x0184 | | dmp_FragmentLightSource[4].position | FSUNIFORM |
| 0x0185 | | | |
| 0x0186 | | dmp_FragmentLightSource[4].spotDirection | FSUNIFORM |
| 0x0187 | | | |
| 0x0189 | | dmp_FragmentLightSource[4].position,<br>dmp_FragmentLightSource[4].twoSideDiffuse,<br>dmp_FragmentLightSource[4].geomFactor0,<br>dmp_FragmentLightSource[4].geomFactor1 | FSUNIFORM |
| 0x018A | | dmp_FragmentLightSource[4].distanceAttenuationBias | FSUNIFORM |
| 0x018B | | dmp_FragmentLightSource[4].distanceAttenuationScale | FSUNIFORM |
| 0x0190 | | dmp_FragmentMaterial.specular0,<br>dmp_FragmentLightSource[5].specular0 | FSUNIFORM |
| 0x0191 | | dmp_LightEnv.lutEnabledRefl, dmp_FragmentMaterial.specular1,<br>dmp_FragmentLightSource[5].specular1 | FSUNIFORM |
| 0x0192 | | dmp_FragmentMaterial.diffuse,<br>dmp_FragmentLightSource[5].diffuse | FSUNIFORM |
| 0x0193 | | dmp_FragmentMaterial.ambient,<br>dmp_FragmentLightSource[5].ambient | FSUNIFORM |
| 0x0194 | | | |

| | | | |
|---|---|---|---|
| | 8.8.5.3 | dmp_FragmentLightSource[5].position | FSUNIFORM |
| 0x0195 | | | |
| 0x0196 | | dmp_FragmentLightSource[5].spotDirection | FSUNIFORM |
| 0x0197 | | | |
| 0x0199 | | dmp_FragmentLightSource[5].position,<br>dmp_FragmentLightSource[5].twoSideDiffuse,<br>dmp_FragmentLightSource[5].geomFactor0,<br>dmp_FragmentLightSource[5].geomFactor1 | FSUNIFORM |
| 0x019A | | dmp_FragmentLightSource[5].distanceAttenuationBias | FSUNIFORM |
| 0x019B | | dmp_FragmentLightSource[5].distanceAttenuationScale | FSUNIFORM |
| 0x01A0 | 8.8.5.3 | dmp_FragmentMaterial.specular0,<br>dmp_FragmentLightSource[6].specular0 | FSUNIFORM |
| 0x01A1 | | dmp_LightEnv.lutEnabledRefl, dmp_FragmentMaterial.specular1,<br>dmp_FragmentLightSource[6].specular1 | FSUNIFORM |
| 0x01A2 | | dmp_FragmentMaterial.diffuse,<br>dmp_FragmentLightSource[6].diffuse | FSUNIFORM |
| 0x01A3 | | dmp_FragmentMaterial.ambient,<br>dmp_FragmentLightSource[6].ambient | FSUNIFORM |
| 0x01A4 | | dmp_FragmentLightSource[6].position | FSUNIFORM |
| 0x01A5 | | | |
| 0x01A6 | | dmp_FragmentLightSource[6].spotDirection | FSUNIFORM |
| 0x01A7 | | | |
| 0x01A9 | | dmp_FragmentLightSource[6].position,<br>dmp_FragmentLightSource[6].twoSideDiffuse,<br>dmp_FragmentLightSource[6].geomFactor0,<br>dmp_FragmentLightSource[6].geomFactor1 | FSUNIFORM |
| 0x01AA | | dmp_FragmentLightSource[6].distanceAttenuationBias | FSUNIFORM |
| 0x01AB | | dmp_FragmentLightSource[6].distanceAttenuationScale | FSUNIFORM |
| 0x01B0 | 8.8.5.3 | dmp_FragmentMaterial.specular0,<br>dmp_FragmentLightSource[7].specular0 | FSUNIFORM |
| 0x01B1 | | dmp_LightEnv.lutEnabledRefl, dmp_FragmentMaterial.specular1,<br>dmp_FragmentLightSource[7].specular1 | FSUNIFORM |
| 0x01B2 | | dmp_FragmentMaterial.diffuse,<br>dmp_FragmentLightSource[7].diffuse | FSUNIFORM |
| 0x01B3 | | dmp_FragmentMaterial.ambient,<br>dmp_FragmentLightSource[7].ambient | FSUNIFORM |
| 0x01B4 | | dmp_FragmentLightSource[7].position | FSUNIFORM |
| 0x01B5 | | | |
| 0x01B6 | | dmp_FragmentLightSource[7].spotDirection | FSUNIFORM |
| 0x01B7 | | | |
| 0x01B9 | | dmp_FragmentLightSource[7].position,<br>dmp_FragmentLightSource[7].twoSideDiffuse,<br>dmp_FragmentLightSource[7].geomFactor0,<br>dmp_FragmentLightSource[7].geomFactor1 | FSUNIFORM |
| 0x01BA | | dmp_FragmentLightSource[7].distanceAttenuationBias | FSUNIFORM |
| 0x01BB | | dmp_FragmentLightSource[7].distanceAttenuationScale | FSUNIFORM |
| 0x01C0 | 8.8.5.2 | dmp_FragmentLighting.ambient, dmp_FragmentMaterial.ambient,<br>dmp_FragmentMaterial.emission | FSUNIFORM |
| 0x01C2 | 8.8.5.1 | dmp_FragmentLightSource[*i*].enabled (*i* = 0 through 7) | FSUNIFORM |
| | | dmp_LightEnv.invertShadow, dmp_LightEnv.shadowAlpha, | |

| | | | |
|---|---|---|---|
| 0x01C3 | 8.8.5.8 | dmp_LightEnv.shadowPrimary, dmp_LightEnv.shadowSecondary, dmp_LightEnv.shadowSelector | FSUNIFORM |
| | 8.8.5.9 | dmp_LightEnv.bumpMode, dmp_LightEnv.bumpRenorm, dmp_LightEnv.bumpSelector, dmp_LightEnv.clampHighlights, dmp_LightEnv.config, dmp_LightEnv.fresnelSelector | FSUNIFORM |
| 0x01C4 | 8.8.5.3 | dmp_FragmentLightSource[*i*].distanceAttenuationEnabled (*i* = 0 through 7), dmp_FragmentLightSource[*i*].shadowed, dmp_FragmentLightSource[*i*].spotEnabled | FSUNIFORM |
| | 8.8.5.9 | dmp_LightEnv.fresnelSelector, dmp_LightEnv.lutEnabledD0, dmp_LightEnv.lutEnabledD1, dmp_LightEnv.lutEnabledRefl | FSUNIFORM |
| 0x01C5 | 8.8.5.4 | `dmp_FragmentMaterial.sampler {D0,D1,FR,RB,RG,RR}` `dmp_FragmentLightSource[i].sampler{SP,DA}` | LUT |
| 0x01C6 | 8.8.5.1 | `dmp_FragmentLighting.enabled` | FSUNIFORM |
| 0x01C8 through 0x01CF | 8.8.5.4 | Sets data to lookup tables. | LUT |
| 0x01D0 | 8.8.5.5 | dmp_LightEnv.absLutInputD0, dmp_LightEnv.absLutInputD1, dmp_LightEnv.absLutInputSP, dmp_LightEnv.absLutInputFR, dmp_LightEnv.absLutInputRB, dmp_LightEnv.absLutInputRG, dmp_LightEnv.absLutInputRR | FSUNIFORM |
| 0x01D1 | 8.8.5.6 | dmp_LightEnv.lutInputD0, dmp_LightEnv.lutInputD1, dmp_LightEnv.lutInputSP, dmp_LightEnv.lutInputFR, dmp_LightEnv.lutInputRB, dmp_LightEnv.lutInputRG, dmp_LightEnv.lutInputRR | FSUNIFORM |
| 0x01D2 | 8.8.5.7 | dmp_LightEnv.lutScaleD0, dmp_LightEnv.lutScaleD1, dmp_LightEnv.lutScaleSP, dmp_LightEnv.lutScaleFR, dmp_LightEnv.lutScaleRB, dmp_LightEnv.lutScaleRG, dmp_LightEnv.lutScaleRR | FSUNIFORM |
| 0x01D9 | 8.8.5.1 | dmp_FragmentLightSource[*i*].enabled (*i* = 0 through 7) | FSUNIFORM |

## 8.10.3. Register Map for Registers 0x0200 – 0x02FF

**Table 8-73. Register Map (Addresses 0x0200 – 0x02FF)**

| Address | Section | Function / Reserved Uniform | NN_GX_STATE_ |
|---|---|---|---|
| 0x0200 | 8.8.1.9 | `glBufferData` | VERTEX |
| 0x0201 | | `glVertexAttribPointer` | VERTEX |
| 0x0202 | | glEnableVertexAttribArray, glDisableVertexAttribArray, glVertexAttribPointer | VERTEX |
| 0x0203 through 0x0226 | | glBufferData, glVertexAttribPointer | VERTEX |
| 0x0227 | 8.8.1.9 | `glBufferData` | - |
| | 8.8.19 | `glDrawElements` | - |
| 0x0228 | 8.8.19 | glDrawElements, glDrawArrays | - |
| 0x0229 | | `glDrawElements` | SHADERMODE |
| 0x022A | | `glDrawArrays` | - |
| 0x022E | | `glDrawArrays` | - |
| 0x022F | | `glDrawElements` | - |
| 0x0231 | | glDrawElements, glDrawArrays | - |
| | | | |

| | | | |
|---|---|---|---|
| 0x0232 | 8.8.1.8 | glVertexAttribPointer, glVertexAttrib{1234}f, glVertexAttrib{1234}fv | VERTEX |
| 0x0233 through 0x0235 | | Vertex attribute data settings. | VERTEX |
| 0x0238 through 0x023D | 8.8.23 | Command buffer execution. nngxAddJumpCommand, nngxAddSubroutineCommand | - |
| 0x0242 | 8.8.1.6 | Sets the number of vertex attributes to input. | SHADERPROGRAM |
| 0x0244 | 8.8.20 | Settings for the shared processor. | SHADERMODE |
| 0x0245 | 8.8.19 | glDrawElements, glDrawArrays | - |
| 0x024A | 8.8.1.10 | Sets the number of used output registers. | SHADERPROGRAM |
| 0x0251 | 8.8.1.10 | Sets the number of used output registers. | SHADERPROGRAM |
| 0x0252 | 8.8.20.12 | Sets the use of subdivision shaders. | SHADERPROGRAM |
| 0x0253 | 8.8.19 | glDrawElements, glDrawArrays | - |
| 0x0254 | 8.8.20.12 | Sets the use of subdivision shaders. | SHADERPROGRAM |
| 0x025E | 8.8.1.10 | Sets the number of used output registers. | SHADERPROGRAM |
| | 8.8.20.8 | Sets the number of used output registers (geometry shaders). | SHADERPROGRAM |
| | 8.8.19 | glDrawElements, glDrawArrays | - |
| 0x025F | 8.8.19 | `glDrawElements, glDrawArrays` | - |
| 0x0280 | 8.8.20.2 | Boolean register (geometry shaders). | VSUNIFORM SHADERMODE |
| 0x0281 through 0x0284 | 8.8.20.3 | Integer registers (geometry shaders). | VSUNIFORM SHADERMODE |
| 0x0289 | 8.8.20.6 | Setting register for number of vertex inputs (geometry shaders). | SHADERPROGRAM |
| | 8.8.20.12 | Sets use of geometry shaders. | SHADERMODE |
| 0x028A | 8.8.20.5 | Starting address setting register (geometry shaders). | SHADERPROGRAM SHADERMODE |
| 0x028B | 8.8.20.7 | Input register mapping setting registers (geometry shaders). | VERTEX |
| 0x028C | | | |
| 0x028D | 8.8.20.9 | Output register mask setting register (geometry shaders). | SHADERPROGRAM SHADERMODE |
| 0x028F | 8.8.20.4 | Program code setting register (geometry shaders). | SHADERBINARY |
| 0x0290 | 8.8.20.1 | Floating-point constant register (geometry shaders). | SHADERFLOAT VSUNIFORM |
| 0x0291 through 0x0298 | | Floating-point constant loading (geometry shaders). | |
| 0x029B | 8.8.20.4 | Program code loading address (geometry shaders). | SHADERBINARY |
| 0x029C through 0x02A3 | | Program code loading (geometry shaders). | |
| 0x02A5 | 8.8.20.4 | Swizzle pattern loading address (geometry shaders). | SHADERBINARY |
| 0x02A6 through 0x02AD | | Swizzle pattern loading (geometry shaders). | |
| 0x02B0 | 8.8.1.2 | Boolean register. | VSUNIFORM SHADERMODE |
| 0x02B1 through 0x02B4 | 8.8.1.3 | Integer register. | VSUNIFORM SHADERMODE |

| | | | |
|---|---|---|---|
| 0x02B9 | 8.8.1.6 | Setting register for number of vertex inputs. | SHADERPROGRAM<br>SHADERMODE |
| 0x02BA | 8.8.1.5 | Starting address setting register. | SHADERPROGRAM<br>SHADERMODE |
| 0x02BB | 8.8.1.7 | Input register mapping setting registers. | VERTEX |
| 0x02BC | | | |
| 0x02BD | 8.8.1.11 | Output register mask setting register. | SHADERPROGRAM<br>SHADERMODE |
| 0x02BF | 8.8.1.4 | Program code setting register. | SHADERBINARY |
| 0x02C0 | 8.8.1.1 | Floating-point constant register. | SHADERFLOAT<br>VSUNIFORM |
| 0x02C1 through 0x02C8 | | Floating-point constant loading. | |
| 0x02CB | 8.8.1.4 | Program code loading address. | SHADERBINARY |
| 0x02CC through 0x02D3 | | Program code loading. | |
| 0x02D5 | 8.8.1.4 | Swizzle pattern loading address. | SHADERBINARY |
| 0x02D6 through 0x02DD | | Swizzle pattern loading. | |



# 9. Profiling Features

The 3DS GPU includes profiling features to allow you to benchmark and fine-tune your hardware performance.

The following table lists the profiling features.

**Table 9-1. List of Profiling Features**

| Feature | Description |
|---|---|
| Busy Counter | Compares the number of times each module outputs a busy signal within a set interval and determines which module output the most busy signals.<br>This is repeated a specified number of times, and the number of times each module was determined to output the most busy signals is counted. |
| Shader Execution Clock Counter | Counts the number of times each of the four shader processors' program counters have transitioned and the clock cycles they have stalled. |
| Vertex Cache Input Vertex Counter | Counts the number of vertices input into the post-vertex cache. |
| Input/Output Polygon Counter | Counts the number of vertices and polygons input into, and the number of polygons output from, the triangle setup module. |
| Input Fragment Counter | Counts the number of fragments input into the per-fragment operation module. |
| Memory Access Counter | Counts the number of times that memory, such as VRAM or the vertex buffer, has been accessed. |

**Note:** Do not call the profiling feature functions while a command list is executing. Doing so does not cause an error, but it can cause the GPU to perform illegal operations.

## 9.1. Starting and Stopping the Profiling Features

Some of the profiling features must be explicitly started or stopped by calling the following functions from your application.

**Code 9-1. Starting and Stopping the Profiling Features**

```
void nngxStartProfiling(GLenum item);
void nngxStopProfiling(GLenum item);
```

For the *item* parameter, specify one of the profiling feature values defined in the following table. Specifying any value that is not defined in this table causes a GL_ERROR_80A2_DMP error on calls to nngxStartProfiling(), and a GL_ERROR_80A3_DMP error on calls to nngxStopProfiling().

**Table 9-2. Defined Values for Starting and Stopping the Profiling Features**

| Defined Value | Profiling Feature |
|---|---|
| NN_GX_PROFILING_BUSY | Busy Counter. (Make sure that you set the parameters to specify the counter's measurement period.) |
| NN_GX_PROFILING_VERTEX_CACHE | Vertex Cache Input Vertex Counter. (Running this can increase power consumption. Make sure that you stop this counter when not using it.) |

## 9.2. Profiling Feature Parameters

Call the following function to set parameters for some of the profiling features.

**Code 9-2. Setting Parameters for the Profiling Features**

```
void nngxSetProfilingParameter(GLenum pname, GLuint param);
```

The following table shows how the values you can specify for *param* differ depending on the defined value specified for *pname*.

**Table 9-3. List of Profiling Feature Parameters**

| Defined Value Specified for *pname* | Value Specified for *param* |
|---|---|
| NN_GX_PROFILING_BUSY_SAMPLING_TIME | Specify the measurement period for each busy counter measurement (number of GPU clock cycles) as a nonzero 16-bit value. |
| | NN_GX_PROFILING_BUSY_SAMPLING_TIME |

| | |
|---|---|
| NN_GX_PROFILING_BUSY_SAMPLING_TIME_MICRO_SECOND | is specified in units of microseconds.<br>When this value is converted to the number of GPU clock cycles, the converted value must fit into 16 bits. So specify the value in the range from 1 through 244. |
| NN_GX_PROFILING_BUSY_SAMPLING_TIME_NANO_SECOND | NN_GX_PROFILING_BUSY_SAMPLING_TIME is specified in units of nanoseconds.<br>When this value is converted to the number of GPU clock cycles, the converted value must fit into 16 bits. So specify the value in the range from 1 through 244537. |
| NN_GX_PROFILING_BUSY_SAMPLING_COUNT | Specify the number of times to measure the busy counter as a 16-bit value.<br>When specifying 0, the busy counter continues counting until explicitly told to stop (on each occurrence, counting for the time specified by NN_GX_PROFILING_BUSY_SAMPLING_TIME, and repeating until stopped).<br>When specifying any other value, the busy counter stops after being explicitly told to stop, or after (measurement time × measurement count) number of clock ticks has passed since the busy counter started.<br>When a value of 1 or greater is set, and the result is retrieved after the measurements have completed, the total of the busy counter measurement results for all modules equals the total number of measurements. |

**Table 9-4. Errors Generated by the nngxSetProfilingParameter() Function**

| Error | Cause |
|---|---|
| GL_ERROR_80A5_DMP | An invalid value was specified for *pname*. |
| GL_ERROR_80A6_DMP | An invalid value was specified for *param*. |

**Note:** Measured times can be converted to GPU clock cycles using a frequency of 268 MHz (268,000,000 Hz).

**Note:** The parameter initial values are undefined, so you must set values for all relevant parameters.

## 9.3. Getting Profiling Results

Call the following function to get the results of profiling.

**Code 9-3. Function for Getting Profiling Results**

```
void nngxGetProfilingResult(GLenum item, GLuint* result);
```

For the *item* parameter, specify the profiling feature for which to get results. GL_ERROR_80A4_DMP An invalid value was specified for *item*.

The *result* parameter stores the results of the specified profiling feature. Note that the size of the buffer required to hold these results changes depending on the profiling feature specified.

For more information about the profiling features you can specify for *item* and the results stored in *result*, see the following sections.

**Note:** All profiling features that do not require explicit starting and stopping run constantly. These profiling feature counters are reset when the hardware is booted. Consequently, when getting profiling results, use the difference between results obtained at the start of the measurement period and results obtained at the end of this period.

Profiling counters restart at zero when they overflow.

## 9.3.1. Busy Counter

Specify a value of NN_GX_PROFILING_BUSY for *item* to get the results of the busy counter. For *result*, specify an array of GLuint types of the same number of elements as NN_GX_PROFILING_RESULT_BUFSIZE_BUSY.

The busy counter is reset to zero when it is started by a call to the nngxStartProfiling() function. It compares the number of times that busy is output in each period of a set length, and counts the number of times each module is busy the most, until the counter is stopped. In addition, each time you get the measurement results, the counter is reset to zero. As a result, if the profiling function is initiated, and then the results are retrieved before the function has stopped, the results retrieved will be those calculated from the previous time the function was run.

The results for each module are stored as 16-bit values. These results are stored for each module in the following order.

**Table 9-5. Busy Counter Data Storage Order**

| Data | Module |
|---|---|
| result[0] bits [31:16] | Shader processor 0 (shared with the geometry processor) |
| result[0] bits [15:0] | Command buffer and vertex array load module |
| result[1] bits [31:16] | Rasterization module |
| result[1] bits [15:0] | Triangle setup |
| result[2] bits [31:16] | Fragment lighting |
| result[2] bits [15:0] | Texture units |
| result[3] bits [31:16] | Per-fragment operation module |
| result[3] bits [15:0] | Texture combiners |

The busy counter result is the number of times a module output the busy signal the most often, not the number of times the busy signal was output.

When the busy counter measurement is started, the number of times each module outputs a busy signal in each measurement period is compared, and the counter for the module outputting the most busy signals is incremented. This measurement is repeated a specified number of times. If multiple modules are tied for outputting the most number of busy signals in a measurement cycle,

the counter for the module among these that is at the earliest stage is incremented. When the busy signal output count for all modules is zero, the counter for the earliest-stage command buffer or vertex array load module is incremented. For this reason, if the number of measurements is set to one or more and the results are obtained after measurements have completed, the sum of the values for all modules will equal the total number of measurements.

You can assume that the modules with the largest values in the measurement results are acting as the bottleneck for the largest amounts of time. Focus your optimization efforts on these modules to fine-tune performance.

#### 9.3.1.1. Resolving Bottlenecks in Triangle Setup/Rasterization Module

You can analyze the possibility that triangle setup (TS) or the rasterization module (RAS) are becoming bottlenecks from the busy counter result.

TS is affected by the polygon count and RAS is affected by both the polygon count and the number of generated pixels. For this reason, they have the following characteristics.

- As long as the processed polygon count, the polygon coordinates, and the number of generated pixels do not change, it does not matter to the performance of TS and RAS whether the polygons are generated by loading vertex data or using a geometry shader.
- As long as the processed polygon count, the polygon coordinates, and the number of generated pixels do not change, it does not matter to the performance of TS and RAS how many times the draw function is called and rendered.
- Changes in the number of vertex attributes (such as the number of texture coordinates) do not affect the performance of TS and RAS.

If you suspect that TS or RAS may be bottlenecks, check whether performance changes if you insert up to 30 or 40 nop in the vertex shader, or when you enable scissoring on the entire screen. If performance does not change in either case, there is a high probability that TS or RAS are bottlenecks. Reduce the polygon count to reduce the processing load on TS. Some methods to achieve this include using the LOD of the rendered object or clipping the object with the CPU. If the load on TS is reduced, the load on RAS will also be reduced.

### 9.3.2. Shader Execution Clock Counter

To get the results of the shader execution clock counter, specify the defined value `NN_GX_PROFILING_VERTEX_SHADER`*n* (where *n* is a value from `0` through 3 that indicates the processor number), corresponding to the shader processor to get, as the value of *`item`*. For *`result`*, specify an array of `GLuint` types of the same number of elements as `NN_GX_PROFILING_RESULT_BUFSIZE_VERTEX_SHADER`*n* (where *n* is a value from `0` through 3 that indicates the processor number). The geometry processor is shared with shader processor 0.

The shader execution clock counter is reset to zero when the hardware is booted, and it runs constantly.

The results are stored as 32-bit values. The following table shows the storage order and the corresponding information.

**Table 9-6. Shader Execution Clock Counter Data Storage Order**

| Data | Description |
|---|---|
| result[0] | Program counter transition count (same as the number of executed shader assembler commands) |
| result[1] | Clock count for stalls due to shader assembler command dependencies (for NN_GX_PROFILING_VERTEX_SHADER0 only, this also includes instances when the geometry shader is valid and the later modules are busy, causing a vout instruction to delay being issued) |
| result[2] | Clock count for stalls due to address register updates (mova commands issued) |
| result[3] | Clock count for stalls due to status register updates (cmp commands issued) |
| result[4] | Clock count for stalls due to program pre-fetch misses (such as when the program counter makes a nonsequential transition due to a branching command) |

## 9.3.3. Vertex Cache Input Vertex Counter

Specify a value of NN_GX_PROFILING_VERTEX_CACHE for *item* to get the results of the vertex cache input vertex counter. For *result*, specify an array of GLuint types of the same number of elements as NN_GX_PROFILING_RESULT_BUFSIZE_BUSY.

The vertex cache input vertex counter is reset to zero when a render is begun that uses the vertex buffer (writes a 1 to register 0x022F). Consequently, you can only get the measurement results for the last executed render. Compare this against the total number of vertex indices used in rendering to estimate the efficiency of the post-vertex cache. However, the count may be slightly high or low depending on differences in when vertex indices are loaded and on whether later modules are busy, even when rendering the same vertex data. In addition, it is possible to get profiling results as if the counter were running, even without calling the nngxStartProfiling() function to start the counter, but such results will probably be incorrect. Make sure that you call the nngxStartProfiling() function to start measurements to get the correct results.

The results are stored as 32-bit values. The following table shows the storage order and the corresponding information.

**Table 9-7. Vertex Cache Input Vertex Counter Data Storage Order**

| Data | Description |
|---|---|
| result[0] | Count of vertices input into the post-vertex cache |

## 9.3.4. Input/Output Polygon Counter

Specify a value of NN_GX_PROFILING_POLYGON for *item* to get the results of the input polygon counter. For *result*, specify an array of GLuint types of the same number of elements as NN_GX_PROFILING_RESULT_BUFSIZE_BUSY.

The input polygon counter is reset to zero when the hardware is booted, and runs constantly.

The results are stored as 32-bit values. The following table shows the storage order and the corresponding information.

**Table 9-8. Input Polygon Counter Data Storage Order**

| Data | Description |
|---|---|

| | |
|---|---|
| result[0] | Number of vertices input into the triangle setup module |
| result[1] | Number of polygons input into the triangle setup module |
| result[2] | Number of polygons output from the triangle setup module |

The number of output polygons is the value of the input polygon count minus the number of polygons stripped out by clipping and culling. Any polygon that intersects the clipping volume is output as multiple polygons, but it is only counted as one polygon.

## 9.3.5. Input Fragment Counter

Specify a value of `NN_GX_PROFILING_FRAGMENT` for *`item`* to get the results of the input fragment counter. For *`result`*, specify an array of `GLuint` types of the same number of elements as `NN_GX_PROFILING_RESULT_BUFSIZE_FRAGMENT`.

The input fragment counter is reset to zero when the hardware is booted, and runs constantly.

The results are stored as 32-bit values. The following table shows the storage order and the corresponding information.

**Table 9-9. Input Polygon Counter Data Storage Order**

| Data | Description |
|---|---|
| result[0] | Number of fragments input into the per-fragment operation module |

The number of fragments does not include fragments discarded by clipping, or by scissor or early depth tests. Fragments are counted before the alpha, stencil, and depth tests, so these test results have no effect on the count.

## 9.3.6. Memory Access Counter

Specify a value of `NN_GX_PROFILING_MEMORY_ACCESS` for *`item`* to get the results of the memory access counter. For *`result`*, specify an array of `GLuint` types of the same number of elements as `NN_GX_PROFILING_RESULT_BUFSIZE_MEMORY_ACCESS`.

The memory access counter is reset to zero when the hardware is booted, and it runs constantly.

The results are stored as 32-bit values. The following table shows the storage order and the corresponding information.

**Table 9-10. Input Polygon Counter Data Storage Order**

| Data | Description |
|---|---|
| result[0] | GPU reads of VRAM (A channel) |
| result[1] | GPU writes to VRAM (A channel) |
| result[2] | GPU reads of VRAM (B channel) |
| result[3] | GPU writes to VRAM (B channel) |
| result[4] | Reads of command buffers, vertex arrays, and index arrays by modules loading command buffers and vertex arrays |

| | |
|---|---|
| result[5] | Texture unit reads of texture memory |
| result[6] | Per-fragment operation module reads of depth and stencil buffers |
| result[7] | Per-fragment operation module writes to depth and stencil buffers |
| result[8] | Per-fragment operation module reads of color buffer |
| result[9] | Per-fragment operation module writes to color buffer |
| result[10] | Upper LCD controller reads of the display buffer |
| result[11] | Lower LCD controller reads of the display buffer |
| result[12] | Post-transfer module reads (transfers by functions such as `nngxTransferRenderImage()` and `glCopyTexImage2D()`) |
| result[13] | Post-transfer module writes (transfers by functions such as `nngxTransferRenderImage()` and `glCopyTexImage2D()`) |
| result[14] | Memory fill module channel 0 buffer writes (buffer clears using functions like `glClear()`) |
| result[15] | Memory fill module channel 1 buffer writes (buffer clears using functions like `glClear()`) |
| result[16] | CPU reads of VRAM (using functions like `glReadPixels()`) |
| result[17] | CPU (DMA transfer) writes to VRAM (DMA transfers via functions like `nngxAddVramDmaCommand()`) |

CONFIDENTIAL

# 10. Function for Directly Specifying Command Lists

CTR-SDK 4.2 added a function that can directly specify command lists without binding them. Applications must manage objects and buffers. However, you can build a framework with a greater degree of freedom because applications do not have to depend on the library's internal status.

The function that directly specifies the command list is called as a `gx Raw()` function and the original function is called as a `gx()` function.

Except for some functions, the suffix "Raw" is appended to the corresponding `gx()` function, and a pointer specification is added to the `nngxCommandList` structure to specify an argument. Basically, there is no difference from the operation and errors generated with the function.

Note the following when using `gx Raw()` functions.

- `gl()` functions and GD library functions, which depend on the current command list, cannot be used with `gx Raw()` functions.
- The `nngxCommandList` structure cannot be copied and used.
- The operation is not guaranteed when a `gx()` function that uses the command list and a `gx Raw()` function are used together.
- There are no corresponding `gx()` functions when using `gx Raw()` functions for saving, reusing, copying, importing, and exporting command lists, in addition to adding jump and subroutine commands. These functions must all be implemented in the application.

**Table 10-1. gx Raw Functions Corresponding to the gx Function Used for the Command List**

| gx API | gx Raw API |
|---|---|
| `nngxGenCmdlists` | No corresponding function |
| | |

| | |
|---|---|
| nngxBindCmdlist | No corresponding function |
| nngxCmdlistStorage | nngxCmdlistStorageRaw<br>nngxGetCommandRequestSizeRaw |
| nngxDeleteCmdlists | No corresponding function |
| nngxSplitDrawCmdlist | nngxSplitDrawCmdlistRaw |
| nngxFlush3DCommand | No corresponding function |
| nngxFlush3DCommandNoCacheFlush | nngxFlush3DCommandNoCacheFlushRaw |
| nngxFlush3DCommandPartially | nngxFlush3DCommandPartiallyRaw |
| nngxAdd3DCommand | No corresponding function |
| nngxAdd3DCommandNoCacheFlush | nngxAdd3DCommandNoCacheFlushRaw |
| nngxAddJumpCommand | No corresponding function |
| nngxAddSubroutineCommand | No corresponding function |
| nngxMoveCommandbufferPointer | nngxMoveCommandbufferPointerRaw |
| nngxAddB2LTransferCommand | nngxAddB2LTransferCommandRaw |
| nngxAddBlockImageCopyCommand | nngxAddBlockImageCopyCommandRaw |
| nngxAddL2BTransferCommand | nngxAddL2BTransferCommandRaw |
| nngxAddMemoryFillCommand | nngxAddMemoryFillCommandRaw |
| nngxAddVramDmaCommand | No corresponding function |
| nngxAddVramDmaCommandNoCacheFlush | nngxAddVramDmaCommandNoCacheFlushRaw |
| nngxFilterBlockImage | nngxFilterBlockImageRaw |
| nngxEnableCmdlistCallback | nngxEnableCmdlistCallbackRaw |
| nngxDisableCmdlistCallback | nngxDisableCmdlistCallbackRaw |
| nngxSetCmdlistCallback | nngxSetCmdlistCallbackRaw |
| nngxRunCmdlist<br>nngxRunCmdlistById | nngxRunCmdlistRaw |
| nngxStopCmdlist | Use the same function |
| nngxReserveStopCmdlist | nngxReserveStopCmdlistRaw |
| nngxWaitCmdlistDone | Use the same function |
| nngxSetTimeout | Use the same function |
| nngxGetIsRunning | Use the same function |
| nngxClearCmdlist | nngxClearCmdlistRaw |
| nngxClearFillCmdlist | nngxClearFillCmdlistRaw |
| nngxStartCmdlistSave | No corresponding function |
| nngxStopCmdlistSave | No corresponding function |
| nngxUseSavedCmdlist | No corresponding function |
| nngxUseSavedCmdlistNoCacheFlush | No corresponding function |
| nngxAddCmdlist | No corresponding function |
| nngxCopyCmdlist | No corresponding function |
| nngxExportCmdlist | No corresponding function |
| nngxImportCmdlist | No corresponding function |
| nngxSetGasAutoAccumulationUpdate | nngxSetGasAutoAccumulationUpdateRaw |
| nngxSetCmdlistParameteri | nngxSetGasUpdateRaw |
| | nngxGetIsRunningRaw<br>nngxGetUsedBufferSizeRaw<br>nngxGetUsedRequestCountRaw |

| | |
|---|---|
| nngxGetCmdlistParameteri | nngxGetMaxBufferSizeRaw<br>nngxGetMaxRequestCountRaw<br>nngxGetTopBufferAddrRaw<br>nngxGetRunBufferSizeRaw<br>nngxGetRunRequestCountRaw<br>nngxGetTopRequestAddrRaw<br>nngxGetNextRequestTypeRaw<br>nngxGetNextBufferAddrRaw<br>nngxGetNextBufferSizeRaw<br>nngxGetHWStateRaw<br>nngxGetCurrentBufferAddrRaw |

The following section explains functions that have different methods of use between the gx API and gx Raw API.

## 10.1. Allocating and Destroying Command Lists

The gx Raw API uses different methods from the gx API to allocate the save data region used in command lists and to destroy command lists.

In the gx API, the following procedure is used to allocate the command list.

1. Generate the command list (nngxGenCmdlist() function).
2. Bind the command list object (nngxBindCmdlist() function).
3. Allocate the save data region (3D command buffer, command request buffer) (nngxCmdlistStorage() function).

By contrast, in the gx Raw API, the following procedure is used to allocate the command list.

1. Get the necessary region size for the Command Request queue (nngxGetCommandRequestSizeRaw() function).
2. Allocate the save data region (3D command buffer, command request buffer).
3. Generate the command list object (nngxCmdlistStorageRaw() function).

The function used to allocate the command list is as follows.

**Code 10-1. gx Raw API Function for Allocating Command Lists**

```
GLsizei nngxGetCommandRequestSizeRaw(GLsizei requestcount);
void nngxCmdlistStorageRaw(nngxCommandList* cmdlist,
                           GLsizei bufsize, GLvoid* commandbuffer,
                           GLsizei requestcount, GLvoid* commandrequest);
```

For the *requestcount* parameter, specify the number of Command Requests that can be queued in the command list.

When allocating the 3D command buffer, note that its address and size (*commandbuffer* and *bufsize*) must be 16-byte aligned. Also, the 3D command buffer must be allocated in system memory.

We recommend that the address of the nngxCommandList structure itself (*cmdlist*) and the command request buffer (*command request*) are 4-byte aligned. There are no restrictions on the memory to allocate the buffer.

Take care not to deallocate any of the buffers while using the command list.

For the gx Raw API, there is no function corresponding to nngxDeleteCmdlists that destroys the command list. The application must manage each save data region for command lists, 3D command buffers, and command request buffers.

## 10.2. Setting Parameters

In the nngxSetCmdlistParameteri() function, which sets parameters for the current command list, the argument *pname* is used to specify the type of parameter set. In the gx Raw API, a different function is called to set each type of parameter.

**Table 10-2. Values Specified for pname to Set Parameter and Corresponding gx Raw Function**

| pname Value | gx Raw API | Parameter Type |
|---|---|---|
| NN_GX_CMDLIST_RUN_MODE | No corresponding function | --- |
| NN_GX_CMDLIST_GAS_UPDATE | nngxSetGasUpdateRaw | GLboolean |

## 10.3. Getting Parameters

In the nngxGetCmdlistParameteri() function, which gets the parameters of the current command list, the argument *pname* is used to specify the parameter type to get and receive the parameter as an argument. In the gx Raw API, a different function is called for each type of parameter and, the parameter can be obtained as a return value.

**Table 10-3. Values Specified for pname and Corresponding gx Raw API for Getting Parameters**

| pname Value | gx Raw API | Return Value Type |
|---|---|---|
| NN_GX_CMDLIST_RUN_MODE | No corresponding function | --- |
| NN_GX_CMDLIST_IS_RUNNING | nngxGetIsRunningRaw | GLboolean |
| NN_GX_CMDLIST_USED_BUFSIZE | nngxGetUsedBufferSizeRaw | GLsizei |
| NN_GX_CMDLIST_USED_REQCOUNT | nngxGetUsedRequestCountRaw | GLsizei |
| NN_GX_CMDLIST_MAX_BUFSIZE | nngxGetMaxBufferSizeRaw | GLsizei |
| NN_GX_CMDLIST_MAX_REQCOUNT | nngxGetMaxRequestCountRaw | GLsizei |
| NN_GX_CMDLIST_TOP_BUFADDR | nngxGetTopBufferAddrRaw | GLvoid* |
| NN_GX_CMDLIST_BINDING | No corresponding function | --- |
| NN_GX_CMDLIST_RUN_BUFSIZE | nngxGetRunBufferSizeRaw | GLsizei |
| NN_GX_CMDLIST_RUN_REQCOUNT | nngxGetRunRequestCountRaw | GLsizei |
| NN_GX_CMDLIST_TOP_REQADDR | nngxGetTopRequestAddrRaw | GLvoid* |
| NN_GX_CMDLIST_NEXT_REQTYPE | nngxGetNextRequestTypeRaw | GLenum |
| NN_GX_CMDLIST_NEXT_REQINFO | nngxGetNextBufferAddrRaw<br>nngxGetNextBufferSizeRaw | GLvoid*<br>GLsizei |
| NN_GX_CMDLIST_HW_STATE | nngxGetHWStateRaw | GLbitfield |
| | | |

| | | |
|---|---|---|
| NN_GX_CMDLIST_CURRENT_BUFADDR | nngxGetCurrentBufferAddrRaw | GLvoid* |

Only the nngxGetHWStateRaw() function is without an argument. All other functions are called by passing the pointer to the nngxCommandList structure.



# Revision History

## Version 1.3 (2015-11-05)

### Additions

- 9.3.1.1. Resolving Bottlenecks in Triangle Setup/Rasterization Module

### Changes

- 2.1.4.1. Changing the Comparison Function
    - Added a figure "Example of Workaround for Comparison Function Change Restrictions."
- 8.1. Saving Command Lists
    - Added a description that the start and end of saving a command list may generate dummy commands as padding in the 3D Command Buffer of the command list.
    - Added a figure "Saving Command Lists."
- 8.1.2. Generating Commands
    - Added information about the dependency relationship of NN_GX_STATE_FRAMEBUFFER.
- 8.1.3. Commands for Partial State Updates and Full State Updates
    - Added distance attenuation in fragment lighting to the "Conditions for a Lookup Table to Be Enabled" table.
- 8.2. Reusing Command Lists
    - Added information about the necessity of the application ensuring that the 3D command buffer region has been flushed.

- Added a figure "Difference in Operation of nngxUseSavedCmdlist() due to copycmd."

- 8.4. Exporting Command Lists
  - Revised the description.

- 8.7.2. 3D Command Buffer Specifications
  - Added a description of dummy commands.

- 8.8.5.3. Light Source Setting Registers (0x0140 – 0x01BF, 0x01C4)
  - Added a figure "Bit Layout of Other Setting Registers (0x01C4, 0x0149 and Others)."

- 8.8.5.5. Lookup Table Argument Range Setting Register (0x01D0)
  - Added a figure "Bit Layout of Lookup Table Argument Range Setting Register (0x01D0)."

- 8.8.5.6. Lookup Table Input Values Setting Register (0x01D1)
  - Added a figure "Bit Layout of Lookup Table Input Value Setting Register (0x01D1)."

- 8.8.5.7. Lookup Table Output Value Scale Setting Register (0x01D2)
  - Added a figure "Bit Layout of Scale Value Setting Register (0x01D2) to Apply to the Lookup Table Output Values."

- 8.8.7.2. Shading Lookup Table Setting Registers (0x0123, 0x0124)
  - Revised an error in the bit specification range.

- 8.8.9.2. Shadow Attenuation Factor Setting Register (0x0130)
  - Added a figure "Bit Layout of Shadow Factor Attenuation Setting Register (0x0130)."

- 8.8.9.3. w Buffer Setting Registers (0x004D, 0x004E, 0x006D)
  - Added a figure "Bit Layout of W-Buffer Setting Registers (0x004D, 0x004E, 0x006D)."

- 8.8.9.6. Framebuffer Access Control Setting Registers (0x0112 – 0x0115)
  - Added a figure "Bit Layout of Framebuffer Access Control Setting Registers (0x0112 – 0x0115)."

- 8.8.20.1. Floating-Point Constant Registers (0x02C0, 0x02C1 – 0x02C8)
  - Added a figure "Bit Layout for the Floating-Point Constant Register Index Specifier (0x0290)."

- 8.8.20.2. Boolean Register (0x02B0)
    - Added a figure "Bit Layout of Boolean Register (0x0280)."
- 8.8.20.3. Integer Registers (0x02B1 – 0x02B4)
    - Added a figure "Bit Layout of Integer Registers (0x0281 - 0x0284)."
- 8.8.20.4. Program Code Setting Registers (0x028F, 0x029B – 0x02A3, 0x02A5 – 0x02AD)
    - Added a figure "Bit Layout of Program Code Loading Registers (0x029B – 0x02A3)."
    - Added a figure "Bit Layout of Swizzle Pattern Loading Registers (0x02A5 – 0x02AD)."
- 8.8.20.5. Starting Address Setting Register (0x02BA)
    - Added a figure "Bit Layout of Starting Address Setting Register (0x028A)."
- 8.8.20.7. Input Register Mapping Setting Registers (0x02BB, 0x02BC)
    - Added a figure "Bit Layout of Input Register Mapping Setting Registers (0x028B, 0x028C)."
- 8.8.20.9. Output Register Mask Setting Register (0x02BD)
    - Added a figure "Bit Layout of Output Register Mask Setting Register (0x028D)."
- 8.8.23.2. Repeated Execution of the Same Command Buffer
    - Added a figure that shows the configuration of jump commands, and a detailed description.
- 9.2. Profiling Feature Parameters
    - Added a table showing the errors generated by `nngxSetProfilingParameter()`.

## Version 1.2 (2015-04-28)

### Changes

- 8.8.1.9. Vertex Attribute Array Setting Registers (0x0200 – 0x0227)
    - Added information about padding in load arrays.
- 8.8.11. Depth Test Setting Registers (0x0107, 0x0126)
    - Corrected information about depth test setting registers.

- 8.8.19.2. Settings When Vertex Buffers Are Not Used
    - Corrected a description of setting registers when vertex buffers are used for rendering.

## Version 1.1 (2015-01-15)

### Changes

- 6.2.3.4. Calculating the Left and Right Camera Matrices
    - Modified the descriptions of functions that perform matrix calculations.
    - Added the description of the function that updates the 3D depth slider value when calculating a matrix.

## Version 1.0 (2014-09-04)

### Additions and Changes

- Initial version.

# 3DS Programming Manual: GD Library

Version 1.1

**Nintendo Confidential**

This document contains confidential and proprietary information of Nintendo, and is protected under confidentiality agreements as well as the intellectual property laws of the United States and of other countries. No part of this document may be released, distributed, transmitted, or reproduced in any form , including by any electronic or mechanical means and by including within information storage and retrieval systems, without written permission from Nintendo.

©2015–2016 Nintendo. All rights reserved.

All company and product names in this document are the trademarks or registered trademarks of their respective companies.

CTR-06-0174-002-D

# 1. Introduction

This document describes programming using the CTR-SDK's GD library. It assumes that the reader is already familiar with basic graphics concepts and how to use some of the libraries (such as the GX library), and focuses specifically on the GD library.

This document also assumes that the reader has an understanding of the content in the *3DS Programming Manual: Basic Graphics* and the *CTR Programming Manual: Advanced Graphics*. Please read those documents first.

## 1.1. The GD Library

The GD library is one of the CTR-SDK graphics libraries. It is mainly used to generate 3D commands.

The library generates only the 3D commands necessary to synchronize its internal state with the GPU state, allowing you to implement graphics processing without keeping track of the GPU state. The API is structured in a way that is tightly linked with the 3DS graphics hardware. It has been designed to achieve high performance while at the same time making it easy for applications to use.

The GD library is incompatible with DMPGL (`gl()` functions). You must use the GX library (`nngx()`

functions) for command list management and some other system-related features. In addition, only a C++ interface is provided.

## 1.1.1. Command Lists

As already mentioned, the GD library does not manage command list operations; the features provided by the GX library are used for that. In other words, applications are solely responsible for implementing ways of preparing, using, and synchronizing command lists.

> **Notes:** The function issuing command requests can only be called in the application core (Core 0).

## 1.1.2. Immediate and Non-Immediate Functions

GD library functions are classified as either immediate or non-immediate functions, depending on when they write 3D commands to the 3D command buffer specified by the current command list object.

*Immediate functions* write data directly to the 3D command buffer when they are called. This has the advantage of allowing you to directly edit recorded packets when you use the packet recording feature (see 6. Packet Recording).

*Non-immediate functions* update the internal state of the GD library without writing to the 3D command buffer. These internal states are loaded when rendering functions (`nn::gd::System::Draw` or `nn::gd::System::DrawIndexed`) are called and data is written to the 3D command buffer as necessary.

Some functions have the characteristics of both immediate and non-immediate functions and update the GD library's internal state while continuing to write to the 3D command buffer.

## 1.1.3. Error Handling

Functions that may fail internally are implemented to return an `nnResult` object. In other words, when an error occurs a corresponding error value is returned.

There are two ways to handle errors with the GD library. The first is to always check the return values from GD functions. The second is to call the `nn::gd::System::SetCallbackFunctionError()` function to register a callback function for handling errors. You can use both of these methods at the same time, getting errors from the return values of the GD functions after the callback is invoked.

**Code 1-1. Setting Callback Functions for Error Handling**

```
typedef void (*nn::gd::System::gdCallbackfunctionErrorPtr)(
            nnResult result, const char* functionName);
static void nn::gd::System::SetCallbackFunctionError(
            nn::gd::System::gdCallbackfunctionErrorPtr callbackFunctionError);
```

**Warning:** The build type (Debug/Development/Release) determines the granularity of error checking within GD functions.

You can also use the `nn::gd::System::GetErrorStringFromResult()` function for debugging purposes to get a message that corresponds to the error for a given `nnResult` object.

**Code 1-2. Getting an Error Message (Only for Debugging)**

```
static char* nn::gd::System::GetErrorStringFromResult(nnResult result);
```

## 1.2. Module

Modules bundle related processing within the 3DS rendering pipeline.

The following modules are defined by the GD library.

**Table 1-1. List of Modules**

| Module | Definition Name |
|---|---|
| Vertex Input Stage | `nn::gd::System::MODULE_VERTEX_INPUT` |
| Shader Stage | `nn::gd::System::MODULE_SHADER` |
| Rasterizer Stage | `nn::gd::System::MODULE_RASTERIZER` |
| Texture Stage | `nn::gd::System::MODULE_TEXTURE` |
| Procedural Texture Stage | `nn::gd::System::MODULE_TEXTURE_PROCEDURAL` |
| Lighting Stage | `nn::gd::System::MODULE_LIGHTING` |
| Combiner Stage | `nn::gd::System::MODULE_TEXTURE_COMBINER` |
| Fog Stage | `nn::gd::System::MODULE_GAS_FOG` |
| Output Stage | `nn::gd::System::MODULE_OUTPUT` |

**Note:** Pipeline processing is sometimes split into stages. Modules are also sometimes called *stages*.

### 1.2.1. Static Class Definitions

Most GD library functions are accessed through `static` classes defined in each module. These stateless classes only have `static()` functions and are never instantiated. In other words, these classes are simply used to group function definitions for each feature.

Each module has one or more classes, as shown in the following table.

**Table 1-2. Modules and Their Classes**

| Module | Classes | Processing |
|---|---|---|
| | | |

| | | |
|---|---|---|
| Vertex Input Stage | `VertexInputStage` | Vertex buffer input. |
| Shader Stage | `ShaderStage` | Attaching shader programs. |
| Rasterizer Stage | `RasterizerStage` | Culling, clipping, viewport settings, scissoring, and early depth tests. |
| Texture Stage | `TextureStage` | Texture unit settings. |
| Procedural Texture Stage | `ProceduralTextureStage` | Procedural texture settings. |
| Lighting Stage | `LightingStage` | Fragment lighting settings. |
| Combiner Stage | `CombinerStage` | Texture combiner settings. |
| Fog Stage | `FogStage` | Fog and gas settings. |
| Output Stage | `OutputStage` | Depth tests, stencil tests, alpha tests, logical operations, blending, write masks, render target settings, and fragment operation settings. |

### 1.2.2. Managing Module States

The library manages its internal state in several modules. If the internal state does not change for two rendering passes, commands related to those modules are not resent. This prevents unnecessary command packets from being generated and can improve performance. If, however, some GD function call has modified the internal state, the corresponding module is considered to be "dirty" (in other words, the module contains changes). Commands that reference the internal state of any module considered to be dirty are sent the next time rendering occurs.

The library tracks whether modules are dirty and automatically processes them. Accordingly, applications that use the GD library normally do not need to track which modules are dirty. However, sometimes it is advantageous to ensure that all the commands related to a module are re-sent. To that end, the GD library provides the following function to explicitly mark modules as dirty.

**Code 1-3. Function for Explicitly Marking Modules as Dirty**

```
void nn::gd::System::ForceDirty(u32 moduleFlag);
```

Specify the modules to mark as dirty to *`moduleFlag`* using a bitwise OR of the definitions in Table 1-1.

This function is useful when the hardware state has been updated (after prepared command packets are executed, for example) or when another graphics library (such as NintendoWare, DMPGL, and GR) is used, and you want to ensure that the entire GD library state is applied.

## 1.3. State Objects and Resource Objects

State objects are immutable objects that are never modified after they are created. They are typically

created during initialization and then used to set a state when necessary at run time. Although the act of creating an object involves memory allocation and data processing time, after a state is created it can be simply output to the 3D command buffer, to minimize later processing. There is no way to directly access the content of these state objects.

When the GD library receives a request to create a state object, it first determines whether it has already created an object that has the exact same settings. If it finds an object with the same settings, it simply returns a pointer to that object rather than creating a new one.

The following are defined as state objects.

**Table 1-3. State Objects**

| State Object | Stage Used | Bundled Settings |
|---|---|---|
| `CombinerState` | Combiner Stage | Combiner and combiner buffer. |
| `BlendState` | Output Stage | Logical operations and blending. |
| `DepthStencilState` | Output Stage | Depth and stencil tests. |
| `InputLayout` | Vertex Input Stage | Vertex buffers. |
| `SamplerState` | Texture Stage | Texture parameters. |

Apart from state objects, there are also the following resource objects. These are used in the same way in the sense that the necessary processing is almost entirely finished when the objects are created.

**Table 1-4. Resource Objects**

| Resource Objects | Description |
|---|---|
| `ShaderBinary` | Saves and loads shader binaries. |
| `Shader` | Loads floating-point constants into the GPU and sets input and output registers. |
| `ShaderPipeline` | Attaches shader programs to the shader pipeline. |
| `VertexBufferResource` | Maintains vertex buffer data. |
| `Texture2DResource` | Maintains resources that have a width and height, such as textures and color buffers. |
| `Texture2D` | Handles a `Texture2DResource` object as a 2D texture. |
| `TextureCube` | Handles six `Texture2DResource` objects as a cube-map texture. |
| `RenderTarget` | Handles a `Texture2DResource` object as a color buffer. |
| `DepthStencilTarget` | Handles a `Texture2DResource` object as a depth (stencil) buffer. |

**Note:** The memory required to create state and resource objects is allocated as a system buffer (`NN_GX_MEM_SYSTEM`) from device memory (`NN_GX_MEM_FCRAM`) through the allocator specified when the GX library was initialized. We recommend that you consider how much memory is used by this library and leave extra space in device memory.

For more information about how much memory is allocated, see 9.2. Size of Memory Allocated Within Functions.

## 1.3.1. Descriptor Classes

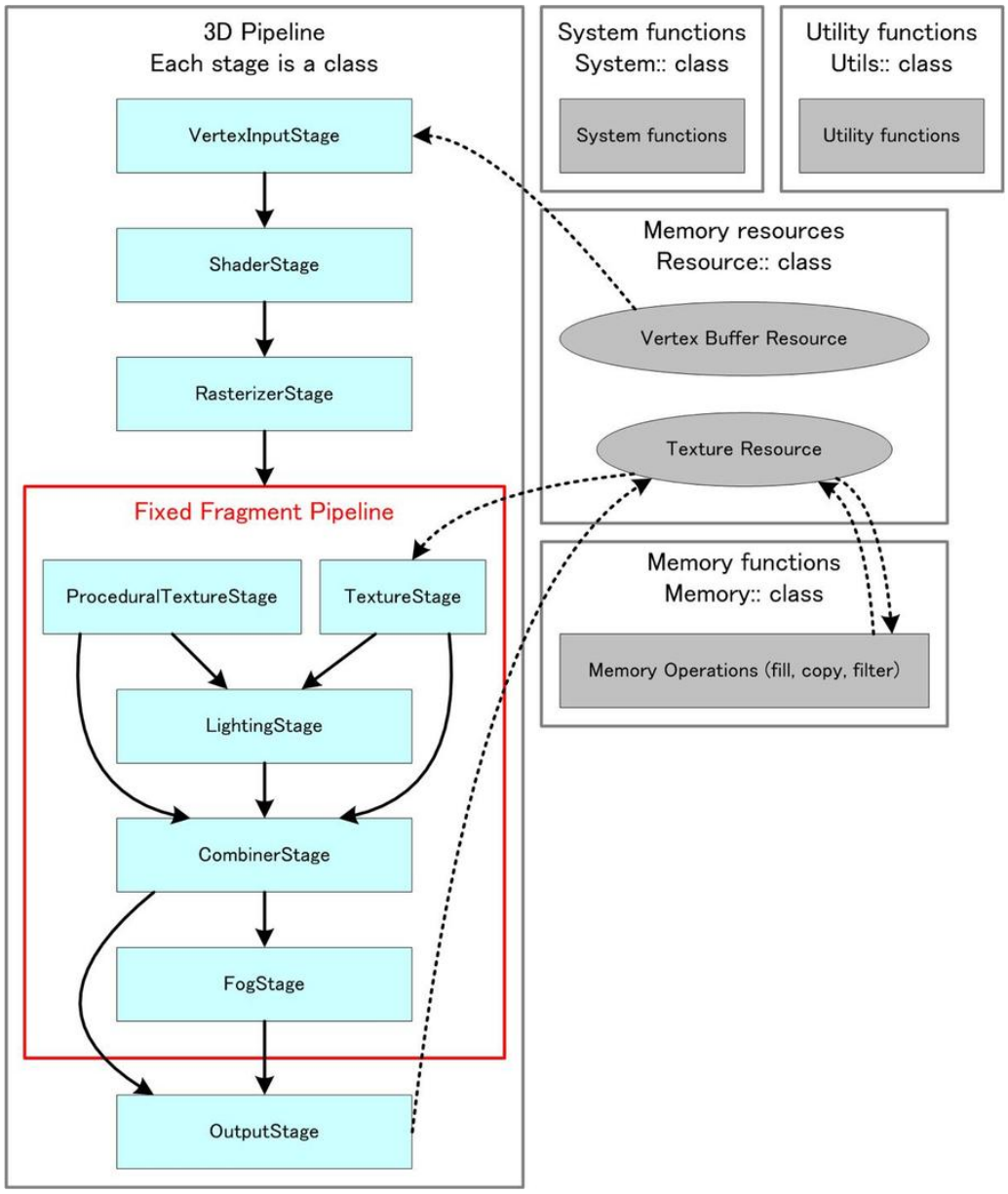
Pass a descriptor class that defines the necessary settings to functions that create objects, such as state and resource objects, which are used to configure the pipeline.

All of a descriptor class's member variables are defined to be `public`. Descriptor classes also sometimes define member functions for easily setting multiple member variables.

## 1.4. Data Flow

The following figure describes each stage of the GD library pipeline from a data perspective and shows how functions are split into classes.

**Figure 1-1. Data Flow in the GD Library**



# 2. Basic Flow of Operations

The following list shows the minimum number and (with slight variations) the order of operations necessary to display graphics using the GD library.

- Initializing the GD Library
- Allocating the Framebuffer
- Loading a Shader Program
- Preparing Vertex Data
- Viewport Settings
- Rendering
- Displaying the Rendered Results
- Finalization

For all other operations, see other chapters, such as 4. Module Descriptions.

**Note:** This chapter does not provide detailed information on each function. For more information, see later chapters in this document or the *CTR-SDK API Reference*.

## 2.1. Initializing the GD Library

The `nn::gd::System::Initialize()` function initializes the GD library. The GD library must be initialized before any other GD functions are called.

The `nn::gd::System::Initialize()` function uses the allocator passed to the `nngxInitialize()` function to allocate internal working memory. You must initialize the GX library using `nngxInitialize()` and allocate a command list for the 3D commands executed within functions before you initialize the GD library.

The following example shows the steps needed to initialize the GD library.

**Code 2-1. Initialization Sample**

```
nngxInitialize(...);

nngxGenCmdlists(1, &cmdListId);
nngxBindCmdlist(cmdListId);
nngxCmdlistStorage(0x80000, 128);
nnResult result = nn::gd::System::Initialize();
```

The GD library can be used in parallel with other frameworks (such as NintendoWare) even when they use the same command lists. However, if command lists are already allocated outside the GD library, do not allocate other command lists until you have initialized the GD library.

The GD library does not have any functions for allocating display buffers. Use `nngx()` functions to allocate display buffers.

# 2.2. Allocating the Framebuffer

Allocate a color buffer and depth (stencil) buffer with the GD library.

Although they have different descriptor class settings, both buffers require the creation of a `Texture2DResource` object. To make it more efficient to clear and render the buffers, we recommend that you allocate the color and depth (stencil) buffers separately in VRAM-A and VRAM-B, as shown in the following sample code.

**Code 2-2. Creating the Texture2DResource Objects Used by the Framebuffer**

```
static nn::gd::Texture2DResource* s_texture2DResource_ColorBuffer = 0;
static nn::gd::Texture2DResource* s_texture2DResource_DepthStencilBuffer = 0;

//Color buffer
nn::gd::Texture2DResourceDescription Text2DResDesc_ColorBuffer =
       {nn::gx::DISPLAY0_WIDTH, nn::gx::DISPLAY0_HEIGHT, 1,
        nn::gd::Resource::NATIVE_FORMAT_RGBA_8888,
        nn::gd::Memory::LAYOUT_BLOCK_8, nn::gd::Memory::VRAMA};
nnResult res = nn::gd::Resource::CreateTexture2DResource(
        &Text2DResDesc_ColorBuffer, 0, GD_FALSE,
        &s_texture2DResource_ColorBuffer);
if ( GD_NNRESULT_IS_FAILURE(res) ) {NN_PANIC("\n");}

//Depth stencil buffer
nn::gd::Texture2DResourceDescription Text2DResDesc_DepthStencilBuffer =
       {nn::gx::DISPLAY0_WIDTH, nn::gx::DISPLAY0_HEIGHT, 1,
        nn::gd::Resource::NATIVE_FORMAT_DEPTH_24_STENCIL_8,
        nn::gd::Memory::LAYOUT_BLOCK_8, nn::gd::Memory::VRAMB};
res = nn::gd::Resource::CreateTexture2DResource(
        &Text2DResDesc_DepthStencilBuffer, 0, GD_FALSE,
        &s_texture2DResource_DepthStencilBuffer);
if ( GD_NNRESULT_IS_FAILURE(res) ) {NN_PANIC("\n");}
```

Using the `Texture2DResource` objects that were created, create a `RenderTarget` object for the color buffer and a `DepthStencilTarget` object for the depth (stencil) buffer.

**Code 2-3. Creating the RenderTarget and DepthStencilTarget Objects**

```
static nn::gd::RenderTarget* s_RenderTarget = 0;
static nn::gd::DepthStencilTarget* s_DepthStencilTarget = 0;

//Color buffer
nn::gd::RenderTargetDescription descRenderTarget = {0};
res = nn::gd::OutputStage::CreateRenderTarget(
        s_texture2DResource_ColorBuffer, &descRenderTarget,
        &s_RenderTarget);
if ( GD_NNRESULT_IS_FAILURE(res) ) {NN_PANIC("\n");}

//Depth stencil buffer
nn::gd::DepthStencilTargetDescription descDepthStencilTarget = {0};
res = nn::gd::OutputStage::CreateDepthStencilTarget(
        s_texture2DResource_DepthStencilBuffer, &descDepthStencilTarget,
        &s_DepthStencilTarget);
if ( GD_NNRESULT_IS_FAILURE(res) ) {NN_PANIC("\n");}
```

Configure the pipeline in the output stage, calling the `SetRenderTarget()` function for the `RenderTarget` object and the `SetDepthStencilTarget()` function for the `DepthStencilTarget` object.

**Code 2-4. Configuring the Pipeline's Framebuffer**

```
//Set the color/depthstencil targets.
nn::gd::OutputStage::SetRenderTarget(s_RenderTarget);
nn::gd::OutputStage::SetDepthStencilTarget(s_DepthStencilTarget);
```

## 2.3. Loading a Shader Program

You must perform the following procedures before applying a shader program to the pipeline so that it can be used for rendering.

- Load a shader binary.
- Create a `ShaderBinary` object.
- Create a `Shader` object.
- Create a `ShaderPipeline` object.
- Apply settings to the pipeline.

The following sample code applies the vertex shader included in a shader binary (with an *rSize*-byte file already loaded in *rBuf*) to the pipeline.

**Code 2-5. Loading a Shader Program**

```
static nn::gd::ShaderBinary* shaderBinary = 0;
static nn::gd::ShaderPipeline* shaderPipeline = 0;
static nn::gd::Shader* vertexShader = 0;

nnResult res;
res = nn::gd::ShaderStage::CreateShaderBinary(rBuf, rSize, &shaderBinary);
if ( GD_NNRESULT_IS_FAILURE(res) ) {NN_PANIC("\n");}
res = nn::gd::ShaderStage::CreateShader(shaderBinary, 0, &vertexShader);
if ( GD_NNRESULT_IS_FAILURE(res) ) {NN_PANIC("\n");}
res = nn::gd::ShaderStage::CreateShaderPipeline(
                           vertexShader, NULL, &shaderPipeline);
if ( GD_NNRESULT_IS_FAILURE(res) ) {NN_PANIC("\n");}

nn::gd::ShaderStage::SetShaderPipeline(shaderPipeline);
```

The register values used by the shader program are set through `UniformLocation` objects that are obtained by specifying a `ShaderPipeline` object and data name as arguments to the `GetShaderUniformLocation()` function in the shader stage.

**Code 2-6. Setting Register Values Using Uniform Settings**

```
static nn::gd::UniformLocation s_shaderVariable_proj;
static nn::gd::UniformLocation s_shaderVariable_view;

s_shaderVariable_proj = nn::gd::ShaderStage::GetShaderUniformLocation(
                           shaderPipeline, "uProjection");
s_shaderVariable_view = nn::gd::ShaderStage::GetShaderUniformLocation(
                           shaderPipeline, "uModelView");
NN_ASSERT(s_shaderVariable_proj.IsValid());
NN_ASSERT(s_shaderVariable_view.IsValid());

nn::math::Matrix44 proj, mv;
nn::gd::ShaderStage::SetShaderPipelineConstantFloat(
        shaderPipeline, s_shaderVariable_proj, static_cast<f32*>(proj));
nn::gd::ShaderStage::SetShaderPipelineConstantFloat(
        shaderPipeline, s_shaderVariable_view, static_cast<f32*>(mv));
```

## 2.4. Preparing Vertex Data

The GD library requires the use of vertex buffers to render primitives.

To use a vertex buffer, you must do the following.

- Create an `InputLayout` object.
- Create a `VertexBufferResource` object.
- Apply the settings (the input layout and vertex buffer) to the pipeline.

The following sample code uses vertex coordinates and vertex color as interleaved vertex data.

**Code 2-7. Preparing Vertex Data**

```
u16 idxs[] = {0, 1, 2};
nnResult res;

float coords_color[] = {
     0.5f, 0.0f, 0.f, 1.f, 1.f, 0.0f, 0.0f,
    -0.5f, 0.5f, 0.f, 1.f, 0.f, 1.0f, 0.0f,
    -0.5f,-0.5f, 0.f, 1.f, 0.f, 0.0f, 1.0f,
};

nn::gd::InputElementDescription descs[] =
{
    { 0, "aPosition",
         nn::gd::VertexInputStage::STREAM_TYPE_FLOAT, 4, 0},
    { 0, "aColor",
         nn::gd::VertexInputStage::STREAM_TYPE_FLOAT, 3, sizeof(float) * 4},
};
u32 strides[] = { sizeof(float) * 7 };
res = nn::gd::VertexInputStage::CreateInputLayout(
        descs, 2, strides, vertexShader, &inputLayout);
if ( GD_NNRESULT_IS_FAILURE(res) ) {NN_PANIC("\n");}

nn::gd::VertexBufferResourceDescription desc;
desc.m_ByteSize = sizeof(coords_color);
desc.m_MemLocation = nn::gd::Memory::FCRAM;
res = nn::gd::Resource::CreateVertexBufferResource(
        &desc, coords_color, &bufferCoord);
if ( GD_NNRESULT_IS_FAILURE(res) ) {NN_PANIC("\n");}

desc.m_ByteSize = sizeof(idxs);
res = nn::gd::Resource::CreateVertexBufferResource(&desc, idxs, &bufferIndex);
if ( GD_NNRESULT_IS_FAILURE(res) ) {NN_PANIC("\n");}

nn::gd::VertexBufferResource* buffers[] = { bufferCoord };
u32 offsets[] = { 0 };
nn::gd::VertexInputStage::SetVertexBuffers(0, 1, buffers, offsets);
nn::gd::VertexInputStage::SetInputLayout(inputLayout);
nn::gd::VertexInputStage::SetIndexBuffer(
        bufferIndex, nn::gd::VertexInputStage::INDEX_FORMAT_USHORT, 0);
```

## 2.5. Configuring the Viewport

Use the `SetViewport()` function in the rasterizer stage to configure the viewport.

The following sample code configures the viewport using information about the render target obtained by calling the `GetRenderTargetProperties()` function in the output stage.

**Code 2-8. Configuring the Viewport**

```
nn::gd::TargetProperties renderTargetProperty;
nn::gd::OutputStage::GetRenderTargetProperties(
        s_RenderTarget, &renderTargetProperty);
nn::gd::Viewport viewPort(0, 0,
        renderTargetProperty.m_Width, renderTargetProperty.m_Height);
nn::gd::RasterizerStage::SetViewport(viewPort);
```

## 2.6. Rendering

You call different rendering functions depending on whether you use vertex indices. Call the `nn::gd::System::DrawIndexed()` function when you use vertex indices and the `nn::gd::System::Draw()` function when you do not. The library also provides `nn::gd::System::DrawImmediate()` and `nn::gd::System::DrawImmediateIndexed()` for rendering without using a vertex buffer resource. Although it is more flexible to handle vertex data if you render without using a vertex buffer resource, rendering is performed more slowly.

Use the `nn::gd::Memory::ClearTargets()` function to clear the framebuffer.

The following sample code clears the framebuffer and then uses vertex indices for rendering.

**Code 2-9. Rendering**

```
//Clear the render buffers.
u8 clearColor[] = {25, 25, 122, 255};
nn::gd::Memory::ClearTargets(s_RenderTarget, s_DepthStencilTarget,
                             clearColor, 1.f, 0);
//Draw.
nn::gd::System::DrawIndexed(3, 0);
```

## 2.7. Displaying the Rendered Results

Use the `nn::gd::Memory::CopyTexture2DResourceBlockToLinear()` function to copy the content of the color buffer to the display buffer and then display the rendered results on the LCDs. This function has nearly the same functionality as the `nngxTransferRenderImage()` function, but the application must use a function such as `nngxGetDisplaybufferParameteri` to get the copy destination address.

**Code 2-10. Displaying the Rendered Results**

```
//Transfer the framebuffer to the display buffer.
int dstAddr;
nngxActiveDisplay(NN_GX_DISPLAY0);
nngxGetDisplaybufferParameteri(NN_GX_DISPLAYBUFFER_ADDRESS, &dstAddr);
nn::gd::Memory::CopyTexture2DResourceBlockToLinear(
    s_texture2DResource_ColorBuffer,            //source
    0,                                          //sourceMipLevelIndex
    0,                                          //srcOffsetY
    (u8*)dstAddr,                               //dstAddr
    nn::gx::DISPLAY0_WIDTH,                     //dstWidth
    nn::gx::DISPLAY0_HEIGHT,                    //dstHeight
    nn::gd::Resource::NATIVE_FORMAT_RGB_888,    //dstFormat
```

```
    nn::gd::Memory::DOWNSCALING_NONE,           //DownScalingMode
    GD_FALSE                                    //yFlip
);
nngxWaitCmdlistDone();
nngxSwapBuffers(NN_GX_DISPLAY_BOTH);
```

## 2.8. Finalizing

The `nn::gd::System::Finalize()` function shuts down the GD library. You can no longer call GD functions after this function has completed.

Finalization automatically deallocates memory, including resources managed by the library and those allocated by the application. However, when possible, have applications explicitly deallocate objects that they create (such as state and resource objects).



# 3. Resource Objects

In general, the `nn::gd::Resource` class defines functions that handle resource objects.

The `nn::gd::Resource` class can create and release resource objects. It can also get detailed information and data pointers for overwriting data.

## 3.1. Texture 2D Resources

Texture 2D resources are resources that have a width and height, such as texture images and color buffers. These resources are handled by `Texture2DResource` objects.

### 3.1.1. Creating a Texture2DResource Object

You can use the `nn::gd::Resource::CreateTexture2DResource()` function to create `Texture2DResource` objects.

**Code 3-1. Creating a Texture2DResource Object**

```
static nnResult nn::gd::Resource::CreateTexture2DResource(
        const nn::gd::Texture2DResourceDescription* description,
        const void* initialData, gdBool autoGenerateMipMapsFromData,
        nn::gd::Texture2DResource** texture2DResource,
        gdBool copyInitialData = GD_TRUE);
```

Set resource information in a descriptor object (`Texture2DResourceDescription`) specified by

the *description* parameter. Further details on descriptor class settings are provided in 3.1.1.2. Descriptor Class for Texture2DResource Objects.

If you are creating a resource that has initial data, such as texture data loaded from a file, specify the starting address of that data in the *initialData* parameter. If you specify GD_TRUE for the *copyInitialData* parameter, the initial data is copied to the memory specified by the descriptor class when the resource is created. Because this function adds a DMA transfer command request to the command list when the resource data is allocated in VRAM, do not overwrite or deallocate the memory region that contains the initial data until the command list finishes executing. If you specify GD_FALSE for *copyInitialData*, the address specified in *initialData* is used as the resource. Note that if a resource contains mipmap data, you must allocate enough memory to store all of it. You also need to pay attention to memory alignment. Applications are responsible for deallocating resource memory.

Specify NULL for *initialData* when you create resources that do not have initial data. A memory region large enough to hold the resource data will be allocated from the memory specified in the descriptor class.

Specify GD_TRUE for the *autoGenerateMipMapsFromData* parameter if you want texture mipmaps to be automatically generated from the initial data. For more information about generating mipmaps automatically, see 3.1.7. Generating Mipmaps Automatically.

Use the pointer that the function sets in *texture2DResource* to access the Texture2DResource object that you have created.

### 3.1.1.1. Helper Functions for Converting Textures Into the Native Format

You must specify initial data in the native 3DS format when you create Texture2DResource objects to be pasted as textures on 3D models. The nn::gd::Resource::Helper class provides helper functions for converting standard textures, such as those used by OpenGL, into the native 3DS format. However, these helper functions have a high processing load and are not well-suited for runtime processing.

There are different helper functions for converting compressed and uncompressed textures, but with the exception of the source pixel format they accept the same arguments.

**Code 3-2. Helper Functions for Converting Textures Into Native Format**

```
static nnResult nn::gd::Resource::Helper::ConvertTextureResourceToNativeFormat(
        nn::gd::Resource::Format format,
        u32 width, u32 height, const u8* dataSrc, u8* dataDst,
        nn::gd::Resource::NativeFormat* pnativeFormat = NULL);
static nnResult
nn::gd::Resource::Helper::ConvertCompressedTextureResourceToNativeFormat(
        nn::gd::Resource::CompressedFormat format,
        u32 width, u32 height, u8* dataSrc, u8* dataDst,
        nn::gd::Resource::NativeFormat* pnativeFormat = NULL);
```

Specify the original pixel format in the *format* parameter.

You can specify the following pixel formats to the ConvertTextureResourceToNativeFormat() function.

- FORMAT_RGBA_8888
- FORMAT_RGB_888
- FORMAT_RGBA_5551

- FORMAT_RGB_565
- FORMAT_RGBA_4444
- FORMAT_LUMINANCE_ALPHA_88
- FORMAT_HILO_88
- FORMAT_LUMINANCE_8
- FORMAT_ALPHA_8
- FORMAT_LUMINANCE_ALPHA_44
- FORMAT_LUMINANCE_4
- FORMAT_ALPHA_4

You can specify the following pixel format in the `ConvertCompressedTextureResourceToNativeFormat()` function.

- FORMAT_ETC1_RGB8

Depending on the pixel format, a buffer large enough to hold the data to convert is temporarily allocated from device memory. A temporary buffer is always allocated when the same address is specified for both *`dataSrc`* and *`dataDst`*.

For more information about the differences between the standard and native texture formats, see the *3DS Programming Manual: Basic Graphics*.

## 3.1.1.2. Descriptor Class for Texture2DResource Objects

This section introduces the member variables of the descriptor class (`Texture2DResourceDescription`) for `Texture2DResource` objects, and describes setting-related cautions and information.

**Code 3-3. Texture2DResourceDescription Class Definition**

```
class nn::gd::Texture2DResourceDescription
{
    u32 m_Width;
    u32 m_Height;
    u32 m_CountMipLevels;
    nn::gd::Resource::NativeFormat m_Format;
    nn::gd::Memory::MemoryLayout m_MemLayout;
    nn::gd::Memory::MemoryLocation m_MemLocation;
};
```

### Width (m_Width)

Sets the width of the resource data in pixels. Specify the width of the top-level texture data for any texture that has mipmaps.

### Height (m_Height)

Sets the height of the resource data in pixels. Specify the height of the top-level texture data for any texture that has mipmaps.

### Mipmap Levels (m_CountMipLevels)

Sets the number of mipmap levels in resource data that has mipmaps. Specify a value of 1 for any texture that does not have mipmaps. Specifying a value of `0` is the same as specifying the maximum number of mipmap levels possible for the resource data's width and height.

### Pixel Format (m_Format)

The pixel formats that you can specify for a resource depend upon how you plan to use it.

The following table shows which pixel formats can be specified for texture data (`Texture2D` or `TextureCube` objects), color buffers (`RenderTarget` objects), and depth stencil buffers (`DepthStencilTarget` objects).

**Table 3-1. Specifiable Pixel Formats**

| Pixel Formats | Texture2D | TextureCube | RenderTarget | DepthStenci |
|---|---|---|---|---|
| `Resource::NATIVE_FORMAT_RGBA_8888` [1] | ○ | ○ | ○ | Error |
| `Resource::NATIVE_FORMAT_RGB_888` [1] | ○ | ○ | Error | Error |
| `Resource::NATIVE_FORMAT_RGBA_5551` [1] | ○ | ○ | ○ | Error |
| `Resource::NATIVE_FORMAT_RGB_565` [1] | ○ | ○ | ○ | Error |
| `Resource::NATIVE_FORMAT_RGBA_4444` [1] | ○ | ○ | ○ | Error |
| `Resource::NATIVE_FORMAT_LUMINANCE_ALPHA_88` | ○ | ○ | Error | Error |
| `Resource::NATIVE_FORMAT_HILO_88` | ○ | ○ | Error | Error |
| `Resource::NATIVE_FORMAT_LUMINANCE_8` | ○ | ○ | Error | Error |
| `Resource::NATIVE_FORMAT_ALPHA_8` | ○ | ○ | Error | Error |
| `Resource::NATIVE_FORMAT_LUMINANCE_ALPHA_44` | ○ | ○ | Error | Error |
| `Resource::NATIVE_FORMAT_LUMINANCE_4` | ○ | ○ | Error | Error |
| `Resource::NATIVE_FORMAT_ALPHA_4` | ○ | ○ | Error | Error |
| `Resource::NATIVE_FORMAT_ETC1_RGB8` | ○ | ○ | Error | Error |
| `Resource::NATIVE_FORMAT_ETC1_A4` | ○ | ○ | Error | Error |
| `Resource::NATIVE_FORMAT_GAS` [2] | ○ | Error | ○ | Error |
| `Resource::NATIVE_FORMAT_SHADOW` [2] | ○ | ○ | ○ | Error |
| `Resource::NATIVE_FORMAT_DEPTH_16` | ○ | ○ | Error | ○ |
| `Resource::NATIVE_FORMAT_DEPTH_24` | ○ | ○ | Error | ○ |
| `Resource::NATIVE_FORMAT_DEPTH_24_STENCIL_8` | ○ | ○ | Error | ○ |

**Note 1:** Supports automatic mipmap generation.
**Note 2:** Does not support mipmaps.

### Memory Layout (m_MemLayout)

You must specify `Memory::LAYOUT_BLOCK_8` as the memory layout for (1) resource objects that are used to create `Texture2D` and `TextureCube` objects, or (2) resource objects that you create with initial data specified.

You can specify `Memory::LAYOUT_BLOCK_8` or `Memory::LAYOUT_BLOCK_32` as the memory

layout for resource objects that are used to create RenderTarget and DepthStencilTarget objects.

You can specify Memory::LAYOUT_LINEAR as the memory layout for any other resource objects.

#### Memory to Allocate for Resource Data (m_MemLocation)

Specifies the memory that will hold the resource data. You cannot create a RenderTarget or DepthStencilTarget object using a resource object created with Memory::FCRAM (device memory) specified.

**Table 3-2. Memory That Can Be Specified for Allocating Resource Data**

| Memory Used to Hold Resource Data | Texture2D | TextureCube | RenderTarget | DepthStencilTarget |
|---|---|---|---|---|
| Memory::FCRAM (device memory) | ○ | ○ | Error | Error |
| Memory::VRAMA (VRAM-A) | ○ | ○ | ○ | ○ |
| Memory::VRAMB (VRAM-B) | ○ | ○ | ○ | ○ |

## 3.1.2. Releasing a Texture2DResource Object

Call the nn::gd::Resource::ReleaseTexture2DResource() function to release a Texture2DResource object. The memory region allocated to create the object is deallocated when the object is released.

**Code 3-4. Releasing a Texture2DResource Object**

```
static nnResult nn::gd::Resource::ReleaseTexture2DResource(
        nn::gd::Texture2DResource* texture2DResource);
```

## 3.1.3. Getting Detailed Information About a Texture2DResource Object

You can use the nn::gd::Resource::GetTexture2DResourceProperties() function to get detailed information (using the Texture2DResourceProperties class) stored in a Texture2DResource object.

**Code 3-5. Getting Detailed Information About a Texture2DResource Object**

```
static nnResult nn::gd::Resource::GetTexture2DResourceProperties(
        const nn::gd::Texture2DResource* texture2DResource,
        nn::gd::Texture2DResourceProperties* properties);
```

When the function returns, the *properties* parameter contains a pointer to the detailed information stored in the Texture2DResource object that was specified in the *texture2DResource* parameter. Ensure that you do not modify the content of the Texture2DResource object through the pointer that is returned.

## 3.1.3.1. Detailed Information About a Texture2DResource Object

An object's detailed information (the `Texture2DResourceProperties` class) has nearly the same settings as the descriptor class that was specified when the object was created.

**Code 3-6. Texture2DResourceProperties Class Definition**

```
class nn::gd::Texture2DResourceProperties
{
    u32 m_Width;
    u32 m_Height;
    u32 m_CountMipLevels;
    u32 m_PixelSize;
    nn::gd::Resource::NativeFormat m_Format;
    nn::gd::Memory::MemoryLayout m_MemLayout;
    nn::gd::Memory::MemoryLocation m_MemLocation;
    u8* m_MemAddr;

    nn::gd::MipmapResourceInfo GetMipmapAddress(u32 mipmapLevel);
};
```

The following member variables include only those that could differ from settings in the descriptor class or those that exist only in this class.

### Mipmap Levels (m_CountMipLevels)

If a value of 0 is specified in the descriptor, this variable is configured with the maximum number of mipmap levels possible for the resource data's width and height.

### Pixel Size (m_PixelSize)

This variable is configured with the number of bits in the resource data's pixel format.

### Starting Address of Resource Data (m_MemAddr)

This variable specifies the physical starting address of the resource data.

### Getting Address Information for Mipmap Data (GetMipMapAddress)

This variable defines a helper function that, given a mipmap level, gets address information (using the `MipmapResourceInfo` class) for the mipmap data.

**Code 3-7. Definition of Address Information for Mipmap Data**

```
class nn::gd::MipmapResourceInfo
{
    u32 m_Width;
    u32 m_Height;
    u8* m_MemAddr;
};
```

This variable defines the width (`m_Width`), height (`m_Height`), and the data's starting address (`m_MemAddr`) of the mipmap data.

## 3.1.4. Getting a Data Pointer in a Texture2DResource Object

You can use the `nn::gd::Resource::MapTexture2DResource()` function to get a data pointer for accessing resource data.

**Code 3-8. Getting a Pointer to the Data in a Texture2DResource Object**

```
static nnResult nn::gd::Resource::MapTexture2DResource(
        nn::gd::Texture2DResource* texture2DResource,
        s32 mipLevelIndex, nn::gd::Resource::MapUsage usage, void** data);
static nnResult nn::gd::Resource::UnmapTexture2DResource(
        nn::gd::Texture2DResource* texture2DResource);
```

This function returns a memory address in the *data* parameter that indicates the starting address of the texture data, at the mipmap level specified by the *mipLevelIndex* parameter, within the resource data of the `Texture2DResource` object specified by the *texture2DResource* parameter. Specify the following mapping methods (types of resource data access) in *usage*.

**Table 3-3. Data Pointer Mapping Methods**

| Definition | Mapping Method | Device Memory | VRAM |
|---|---|---|---|
| `Resource::MAP_READ_ONLY` | Read-only | ○ | ○ |
| `Resource::MAP_WRITE_ONLY` | Write-only | ○ | Error |
| `Resource::MAP_READ_WRITE` | Read/write | ○ | Error |

**Note:** The mapping method is only checked when you get a data pointer. Although writing to the resource data in device memory obtained by `MAP_READ_ONLY` does not result in an error, the cache is not flushed when the mapping is removed.

You cannot directly access resource data in VRAM. Use a DMA transfer or some other means to copy the data to memory that can be directly accessed by the CPU.

After you have finished accessing the resource data, remove the mapping using the `nn::gd::Resource::UnmapTexture2DResource()` function. If you get a data pointer to resource data in device memory using a mapping method that includes writes, the cache is flushed when the mapping is removed.

## 3.1.5. Clearing Resource Data (Texture2DResource Objects)

You can clear resource data using a specified value. You can also specify a range of mipmap levels to clear in resources that have mipmap data. However, **you cannot clear resource data allocated in device memory**.

**Code 3-9. Clearing Resource Data (Texture2DResource Objects)**

```
static nnResult nn::gd::Memory::ClearTexture2DResource(
        const nn::gd::Texture2DResource* tex2DResource,
        s32 mipLevelIndexStart, s32 mipLevelIndexEnd, const u8 Components[4]);
static nnResult nn::gd::Memory::ClearTexture2DResource(
        const nn::gd::Texture2DResource* tex2DResource,
        s32 mipLevelIndexStart, s32 mipLevelIndexEnd, u32 value);
```

This function attempts to clear any resource when the *value* parameter is specified, but it is limited to the following pixel formats when the *Components* parameter is specified. The right side of the following figure shows how values specified to *Components* are applied to the data to be cleared.

**Figure 3-1. Clearing Resource Data: Structure and Valid Pixel Formats**

## 3.1.6. Partially Copying Resource Data (Texture2DResource Objects)

You can specify a mipmap level and rectangular region to create a partial copy of resource data. Its format is not converted because this function calls the `nngxAddBlockImageCopyCommand()` function. Data is actually copied when the command list is executed.

**Code 3-10. Partially Copying Resource Data (Texture2DResource Objects)**

```
static nnResult nn::gd::Memory::CopyTextureSubResource(
        const nn::gd::Texture2DResource* source,
        s32 srcMipLevelIndex, const nn::gd::Memory::Rect& rect,
```

```
        const nn::gd::Texture2DResource* dest, s32 dstMipLevelIndex,
        s32 destPosX, s32 destPosY);
```

The resource data and the viewport (LCD) use different origins, which this function takes into account. Specify the rectangular region (*rect*) and the coordinates to copy to (*destPosX* and *destPosY*), based on the viewport's origin.

An error (the `ResultInvalidTextureFormat()` function) is returned when the source and destination have different pixel sizes.

## 3.1.7. Generating Mipmaps Automatically

You can automatically generate mipmap data for `Texture2DResource` objects.

**Code 3-11. Generating Mipmaps Automatically**

```
static nnResult nn::gd::Memory::GenerateMipMaps(
        const nn::gd::Texture2DResource* tex2DResource,
        u32 mipLevelSourceIndex, s32 mipLevelLastDestinationIndex);
```

*The `Texture2DResource` object specified for `tex2DResource`* must be created so that it can hold the resource data generated for the mipmap levels. You can only generate mipmap data automatically for `Texture2DResource` objects created in the following formats.

- `nn::gd::Resource::NATIVE_FORMAT_RGBA_8888`
- `nn::gd::Resource::NATIVE_FORMAT_RGBA_5551`
- `nn::gd::Resource::NATIVE_FORMAT_RGBA_4444`
- `nn::gd::Resource::NATIVE_FORMAT_RGB_888`
- `nn::gd::Resource::NATIVE_FORMAT_RGB_565`

You can use the `nn::gd::Resource::CreateTexture2DResourceCastFrom()` function to convert an object from a format that is not in this list to a format such as `NATIVE_FORMAT_RGBA_8888`, which generates mipmap data. This method is not guaranteed to produce proper mipmap data, however, because pixels comprise different bits in each format.

Mipmap data is automatically generated from the level specified by the *mipLevelSourceIndex* parameter to the level specified by the *mipLevelLastDestinationIndex* parameter. To specify the maximum number of mipmap levels for that resource, specify a value of –1 for *mipLevelLastDestinationIndex*.

### 3.1.7.1. Using the CPU to Generate Mipmaps Automatically

You can also call the `nn::gd::Resource::Helper::GenerateMipMapsCPU()` function to generate mipmap data automatically. This function uses the CPU to generate mipmaps and is slower than `nn::gd::Memory::GenerateMipMaps`, which uses the GPU, but it has the advantage of supporting more formats and allowing you to generate mipmap data with a minimum width and height of 8 pixels. (The minimum width and height is either 32 or 64 pixels, depending on the format, when the GPU is used.)

**Code 3-12. Using the CPU to Generate Mipmaps Automatically**

```
static nnResult nn::gd::Resource::Helper::GenerateMipMapsCPU(
        nn::gd::Resource::NativeFormat format,
        u32 width, u32 height, const u8* dataSrc, u8* dataDst,
        u32 countMipLevelToGenerate);
```

You can specify the following formats for the *format* parameter.

- nn::gd::Resource::NATIVE_FORMAT_RGBA_8888
- nn::gd::Resource::NATIVE_FORMAT_RGBA_5551
- nn::gd::Resource::NATIVE_FORMAT_RGBA_4444
- nn::gd::Resource::NATIVE_FORMAT_RGB_888
- nn::gd::Resource::NATIVE_FORMAT_RGB_565
- nn::gd::Resource::NATIVE_FORMAT_LUMINANCE_ALPHA_88
- nn::gd::Resource::NATIVE_FORMAT_LUMINANCE_ALPHA_44
- nn::gd::Resource::NATIVE_FORMAT_LUMINANCE_8
- nn::gd::Resource::NATIVE_FORMAT_ALPHA_8
- nn::gd::Resource::NATIVE_FORMAT_HILO_88

Specify the width and height of the original resource data in *width* and *height*, respectively. Both the width and height must be at least 8 and a power of 2.

Specify the starting address of the resource data in the *dataSrc* parameter, and specify the address to store the mipmap data in the *dataDst* parameter. Allocate enough memory to store all of the generated mipmap data.

For the *countMipLevelToGenerate* parameter, specify the number of mipmap levels to generate.

## 3.1.8. Converting a Texture2DResource Object to Another Format

You can use the following function to convert an existing Texture2DResource object to another format. This allows the resource data for RenderTarget objects that were used for rendering (such as gas textures, shadow textures, and depth buffers) to be reused as texture data (resource data for Texture2D objects).

**Code 3-13. Converting a Texture2DResource Object to Another Format**

```
static nnResult nn::gd::Resource::CreateTexture2DResourceCastFrom(
        const nn::gd::Texture2DResource* initialTexture2DResource,
        nn::gd::Resource::NativeFormat format,
        nn::gd::Memory::MemoryLayout layout,
        nn::gd::Texture2DResource** texture2DResource);
```

The object returned in the *texture2DResource* parameter has the pixel format and memory layout of the Texture2DResource object specified for the *initialTexture2DResource* parameter changed to the values specified by the *format* and *layout* parameters, respectively. An error is returned if the two pixel formats are not compatible (if they have different pixel sizes).

This function creates a new Texture2DResource object that contains modified format information. However, the data is not actually converted from one format to another, nor is data actually copied.

The original object remains unchanged. All information other than the pixel format and memory layout, such as the location at which data is stored, remains unchanged. If you overwrite resource data using the object created by this function, you also overwrite the resource data of the original object.

# 3.2. Vertex Buffer Resources

*Vertex buffer resources* are vertex coordinates, vertex indices, and other resources that are used as vertex buffers. These resources are handled by `VertexBufferResource` objects.

## 3.2.1. Creating a VertexBufferResource Object

You can use the `nn::gd::Resource::CreateVertexBufferResource()` function to create `VertexBufferResource` objects.

**Code 3-14. Creating a VertexBufferResource Object**

```
static nnResult nn::gd::Resource::CreateVertexBufferResource(
        const nn::gd::VertexBufferResourceDescription* description,
        const void* initialData, nn::gd::VertexBufferResource** buffer,
        gdBool copyInitialData = GD_TRUE);
```

Set resource information in a descriptor object (`VertexBufferResourceDescription`) specified by the *`description`* parameter. For more information about descriptor class settings, see 3.2.1.1. Descriptor Class for VertexBufferResource Objects.

If you are creating a resource that has initial data, such as vertex data loaded from a file, specify the starting address of that data in the *`initialData`* parameter. If you specify `GD_TRUE` for the *`copyInitialData`* parameter, the initial data is copied to the memory specified by the descriptor class when the resource is created. Because this function adds a DMA transfer command request to the command list when the resource data is allocated in VRAM, do not overwrite or deallocate the memory region that contains the initial data until the command list finishes executing. Pay attention to the memory alignment of the address specified in *`initialData`*, because it is used as the resource if you specify `GD_FALSE` for *`copyInitialData`*. Applications are responsible for deallocating resource memory.

Specify `NULL` for *`initialData`* when you create resources that do not have initial data. A memory region large enough to hold the resource data is allocated from the memory specified in the descriptor class.

The function sets a pointer in the *`buffer`* parameter that you can use to access the `VertexBufferResource` that it just created.

### 3.2.1.1. Descriptor Class for VertexBufferResource Objects

This section introduces the member variables of the descriptor class (`VertexBufferResourceDescription`) for `VertexBufferResource` objects and provides setting-related cautions and information.

**Code 3-15. VertexBufferResourceDescription Class Definition**

```
class nn::gd::VertexBufferResourceDescription
{
    u32 m_ByteSize;
    nn::gd::Memory::MemoryLocation m_MemLocation;
};
```

#### Vertex Buffer Size (m_ByteSize)

Sets the number of bytes in the vertex buffer (resource data).

#### Memory to Allocate for Resource Data (m_MemLocation)

Specifies the memory that will hold the resource data.

The memory to be specified can be selected from `Memory::FCRAM` (device memory), `Memory::VRAMA` (VRAM-A), or `Memory::VRAMB` (VRAM-B).

## 3.2.2. Releasing a VertexBufferResource Object

Call the `nn::gd::Resource::ReleaseVertexBufferResource()` function to release a `VertexBufferResource` object. The memory region allocated to create the object is deallocated when the object is released.

**Code 3-16. Releasing a VertexBufferResource Object**

```
static nnResult nn::gd::Resource::ReleaseVertexBufferResource(
        nn::gd::VertexBufferResource *buffer);
```

## 3.2.3. Getting Detailed Information About a VertexBufferResource Object

You can use the `nn::gd::Resource::GetVertexBufferResourceProperties()` function to get detailed information (using the `VertexBufferResourceProperties` class) stored in a `VertexBufferResource` object.

**Code 3-17. Getting Detailed Information About a VertexBufferResource Object**

```
static nnResult nn::gd::Resource::GetVertexBufferResourceProperties(
        const nn::gd::VertexBufferResource* buffer,
        nn::gd::VertexBufferResourceProperties* properties);
```

This function sets a pointer in the *`properties`* parameter that points to the detailed information stored in the `VertexBufferResource` object specified in the *`buffer`* parameter. Ensure that you do not modify the content of the `VertexBufferResource` object using the pointer that you get.

### 3.2.3.1. Detailed Information About a VertexBufferResource Object

An object's detailed information (the `VertexBufferResourceProperties` class) has nearly the same settings as the descriptor class that was specified when the object was created.

**Code 3-18. VertexBufferResourceProperties Class Definition**

```
class nn::gd::VertexBufferResourceProperties
{
    u32 m_ByteSize;
    nn::gd::Memory::MemoryLocation m_MemLocation;
    u8* m_MemAddr;
};
```

The rest of this section describes member variables that only exist in this class.

#### Starting Address of Resource Data (m_MemAddr)

This variable specifies the physical starting address of the resource data.

## 3.2.4. Getting a Pointer to the Data in a VertexBufferResource Object

You can use the `nn::gd::Resource::MapVertexBufferResource()` function to get a data pointer for accessing resource data.

**Code 3-19. Getting a Pointer to the Data in a VertexBufferResource Object**

```
static nnResult nn::gd::Resource::MapVertexBufferResource(
        nn::gd::VertexBufferResource* buffer,
        nn::gd::Resource::MapUsage usage, void** data);
static nnResult nn::gd::Resource::UnmapVertexBufferResource(
        nn::gd::VertexBufferResource* buffer);
```

The starting address of the resource data for the `VertexBufferResource` object specified by the *`buffer`* parameter is returned in the *`data`* parameter. Specify the following mapping methods (types of resource data access) in *`usage`*.

**Table 3-4. Data Pointer Mapping Methods**

| Definition | Mapping Method | Device Memory | VRAM |
|---|---|---|---|
| `Resource::MAP_READ_ONLY` | Read-only | ○ | ○ |
| `Resource::MAP_WRITE_ONLY` | Write-only | ○ | Error |
| `Resource::MAP_READ_WRITE` | Read/write | ○ | Error |

**Note:** The mapping method is only checked when you get a data pointer. Although writing to the resource data in device memory obtained by `MAP_READ_ONLY` does not result in an error, the cache is not flushed when the mapping is removed.

You cannot directly access resource data in VRAM. Use a method such as DMA transfer to copy the data to memory that can be directly accessed by the CPU.

After you have finished accessing the resource data, remove the mapping using the `nn::gd::Resource::UnmapVertexBufferResource()` function. If you get a data pointer to resource data in device memory using a mapping method that includes writes, the cache is flushed when the mapping is removed.

### 3.2.5. Partially Copying Resource Data (VertexBufferResource Objects)

You can create a partial copy of resource data. Data is actually copied when the command list is executed because this function calls the `nngxAddBlockImageCopyCommand()` function.

**Code 3-20. Partially Copying Resource Data (VertexBufferResource Objects)**

```
static nnResult nn::gd::Memory::CopyVertexBufferSubResource(
        const nn::gd::VertexBufferResource* source,
        u32 sourceOffset, u32 size,
        const nn::gd::VertexBufferResource* dest, u32 destOffset);
```

Data is not copied if the arguments passed to the *`sourceOffset`*, *`size`*, and *`destOffset`* parameters are not all multiples of 16.



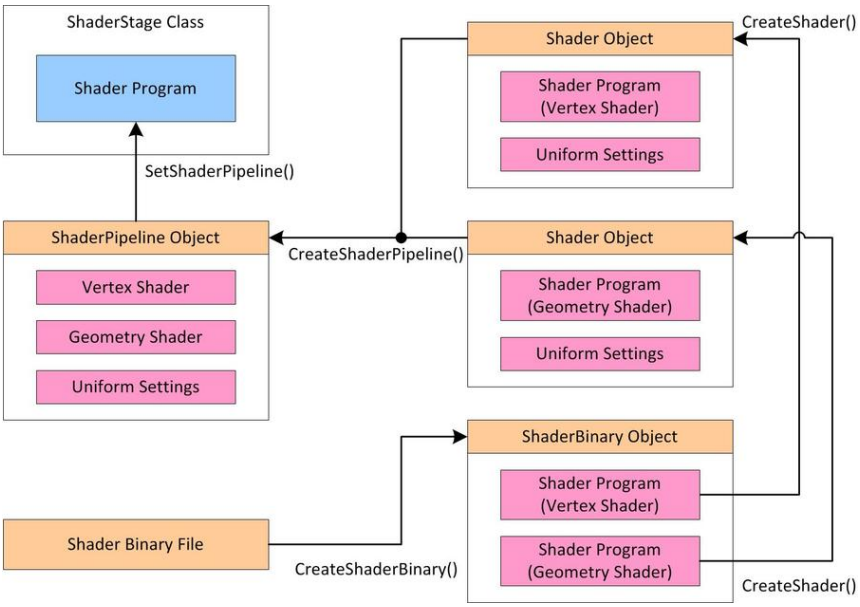
# 4. Module Descriptions

## 4.1. Shader Stage

The vertex shader and geometry shader are configured in the shader stage by functions defined in the `nn::gd::ShaderStage` class.

The following figure provides an overview of the settings made in the shader stage.

**Figure 4-1. Overview of the Shader Stage**




Applications load vertex shaders and geometry shaders as precompiled shader binary files. A shader binary file can contain one or more vertex shader and geometry shader programs.

To access a particular shader in a shader binary file, pass the loaded shader binary file to the `nn::gd::ShaderStage::CreateShaderBinary()` function and then use the `ShaderBinary` object that is generated.

**Code 4-1. Creating a ShaderBinary Object**

```
static nnResult nn::gd::ShaderStage::CreateShaderBinary(
        const void* shaderBytecode, u32 bytecodeLength,
        nn::gd::ShaderBinary** shaderBinary);
```

Specify the starting address of the buffer storing the loaded shader binary code in the *`shaderBytecode`* parameter, and specify the size of the code (in bytes) in the *`bytecodeLength`* parameter.

Use the pointer that you receive from *`shaderBinary`* to access the `ShaderBinary` object that you have created.

Use the `nn::gd::ShaderStage::CreateShader()` function to create a `Shader` object for accessing the shader program contained within the `ShaderBinary` object. A `Shader` object represents the code for a single shader (either a vertex shader or a geometry shader).

**Code 4-2. Creating a Shader Object**

```
static nnResult nn::gd::ShaderStage::CreateShader(
        nn::gd::ShaderBinary* shaderBinary, u32 shaderBinaryIndex,
        nn::gd::Shader** shader);
```

If the `ShaderBinary` object specified in the *`shaderBinary`* parameter has multiple shader programs, specify the shader program index in the *`shaderBinaryIndex`* parameter to create a `Shader` object.

The function sets a pointer in the *`shader`* parameter to the created `Shader` object.

Call the `nn::gd::ShaderStage::CreateShaderPipeline()` function with one or two `Shader` objects to create a `ShaderPipeline` object that you can use to apply the shader programs to the pipeline.

**Code 4-3. Creating a ShaderPipeline Object and Applying Shader Programs to the Pipeline**

```
static nnResult nn::gd::ShaderStage::CreateShaderPipeline(
        nn::gd::Shader* vertexShader, nn::gd::Shader* geometryShader,
        nn::gd::ShaderPipeline** shaderPipeline,
        nn::gd::ShaderPipelineUnmanagedRegisters* unmanagedRegister = NULL);
static nnResult nn::gd::ShaderStage::SetShaderPipeline(
        nn::gd::ShaderPipeline* shaderPipeline);
```

Use the *vertexShader* parameter to specify the Shader object to use as the vertex shader. Use the *geometryShader* parameter to specify the Shader object to use as the geometry shader, or NULL if a geometry shader is not required.

Use the *unmanagedRegister* parameter to specify a range of registers for which the library will not automatically generate commands. For more information about this setting, see 4.1.2. Commands Automatically Generated by the Library to Set Registers.

**Warning:** The GD library does not check whether the vertex shader outputs the proper vertex attributes to the geometry shader. To ensure that the vertex attributes output by the vertex shader match the vertex attributes used as input to the geometry shader, create a ShaderPipeline object with the correct combination of Shader objects.

The following sample code first creates a ShaderPipeline object that only uses a vertex shader, and then applies that object to the pipeline.

**Code 4-4. Configuring the Shader Stage**

```
nn::gd::ShaderBinary* shaderBinary = 0;
nn::gd::ShaderPipeline* shaderPipeline = 0;
nn::gd::Shader* vertexShader = 0;
nn::gd::ShaderStage::CreateShaderBinary(
        shaderBinaryFileBuffer, bufferSize, &shaderBinary);
nn::gd::ShaderStage::CreateShader(shaderBinary, 0, &vertexShader);
nn::gd::ShaderStage::CreateShaderPipeline(vertexShader, NULL, &shaderPipeline);
nn::gd::ShaderStage::SetShaderPipeline(shaderPipeline);
```

## 4.1.1. Configuring Registers for Shader Settings

A ShaderPipeline object maintains values configured for floating-point constant registers and other registers that are used for shader settings. The GD library allows you to use shader stage functions to change register values for shader settings. These functions update variables that are temporarily saved as the international state of the ShaderPipeline object, and are not output to the 3D command buffer until a render command request.

The following table shows which functions to use to set values in different types of registers.

**Table 4-1. Register Types and Functions Called**

| Type of Register | Function |
|---|---|
| Floating-point constant register | nn::gd::ShaderStage::SetShaderPipelineConstantFloat |
| Boolean register | nn::gd::ShaderStage::SetShaderPipelineConstantBoolean |
| Integer register | nn::gd::ShaderStage::SetShaderPipelineConstantInteger |

**Code 4-5. Functions Used to Configure Values to Registers for Shader Settings**

```
static nnResult nn::gd::ShaderStage::SetShaderPipelineConstantFloat(
        nn::gd::ShaderPipeline* shaderPipeline,
        nn::gd::UniformLocation uniformLocation, f32* v);
static nnResult nn::gd::ShaderStage::SetShaderPipelineConstantBoolean(
        nn::gd::ShaderPipeline* shaderPipeline,
        nn::gd::UniformLocation uniformLocation, u16 v, u32 count);
static nnResult nn::gd::ShaderStage::SetShaderPipelineConstantInteger(
        nn::gd::ShaderPipeline* shaderPipeline,
        nn::gd::UniformLocation uniformLocation, u8* v);
static nn::gd::UniformLocation nn::gd::ShaderStage::GetShaderUniformLocation(
        nn::gd::ShaderPipeline* shaderPipeline, const char* name);
```

Get a `UniformLocation` object using the `nn::gd::ShaderStage::GetShaderUniformLocation()` function, and then use the object to specify which register value to change. Use the *name* parameter to specify the name of the uniform setting when getting the object. The `IsValid()` function of the `UniformLocation` object returns a value of 1 when the correct uniform setting has been obtained.

## 4.1.2. Commands Automatically Generated by the Library to Set Registers

Commands to set registers are normally generated automatically when a different `ShaderPipeline` object (from the one currently being applied) is applied to the pipeline. However, performance may be improved in some cases by generating commands that write directly to the registers, rather than having commands generated automatically.

The `nn::gd::ShaderStage::SetFloatConstantBuffer()` function takes this into account and allows you to change register settings independently.

**Code 4-6. Changing Register Settings Independently**

```
static nnResult nn::gd::ShaderStage::SetFloatConstantBuffer(
        nn::gd::ShaderStage::ShaderType shaderType,
        u32 firstRegisterIndex, u32 registerCount, f32* constantBufferSrc);
```

This function can only change the value of *unmanaged* floating-point constant registers, which are not set by automatically generated commands. The range of unmanaged registers is specified by the `ShaderPipelineUnmanagedRegisters` object given as the *`unmanagedRegister`* argument when a `ShaderPipeline` object is created.

The `ShaderPipelineUnmanagedRegisters` class comprises an array of the `UnmanagedRegistersInterval` objects and the number of elements in that array. The `UnmanagedRegistersInterval` class comprises the shader type (`m_ShaderType`), the number of the first unmanaged register (`m_FirstUnmanagedRegister`), and the number of unmanaged registers (`m_RegisterCount`).

**Code 4-7. Classes Used to Specify Unmanaged Registers**

```
class nn::gd::UnmanagedRegistersInterval
{
    nn::gd::ShaderStage::ShaderType m_ShaderType;
    u32 m_FirstUnmanagedRegister;
    u32 m_RegisterCount;
};
class nn::gd::ShaderPipelineUnmanagedRegisters
{
```

```
    nn::gd::UnmanagedRegistersInterval* m_ArrayUnmanagedRegistersInterval;
    u32 m_CountUnmanagedRegistersInterval;
};
```

The following sample code shows how to create a ShaderPipeline object that includes unmanaged registers.

**Code 4-8. Creating a ShaderPipeline Object Including Unmanaged Registers**

```
nn::gd::ShaderPipelineUnmanagedRegisters unmanagedRegs;
nn::gd::UnmanagedRegistersInterval arrayUnmanagedRegistersInterval[] =
{
    {nn::gd::ShaderStage::SHADER_TYPE_VERTEX, 0, 4},
    {nn::gd::ShaderStage::SHADER_TYPE_VERTEX, 4, 3},
    {nn::gd::ShaderStage::SHADER_TYPE_VERTEX, 8, 3}
};

unmanagedRegs.m_ArrayUnmanagedRegistersInterval =
        arrayUnmanagedRegistersInterval;
unmanagedRegs.m_CountUnmanagedRegistersInterval = 3;
nn::gd::ShaderStage::CreateShaderPipeline(
        s_vertexShader, NULL, &s_shaderPipeline, &unmanagedRegs);
```

The following code is an example of writing directly to unmanaged registers.

**Code 4-9. Writing to Unmanaged Registers**

```
nn::gd::UniformLocation s_shaderVariable_proj =
    nn::gd::ShaderStage::GetShaderUniformLocation(
        s_shaderPipeline, "uProjection");

nn::gd::ShaderStage::SetFloatConstantBuffer(
    nn::gd::ShaderStage::SHADER_TYPE_VERTEX,
        s_shaderVariable_proj.getRegister(), 4, data);
```

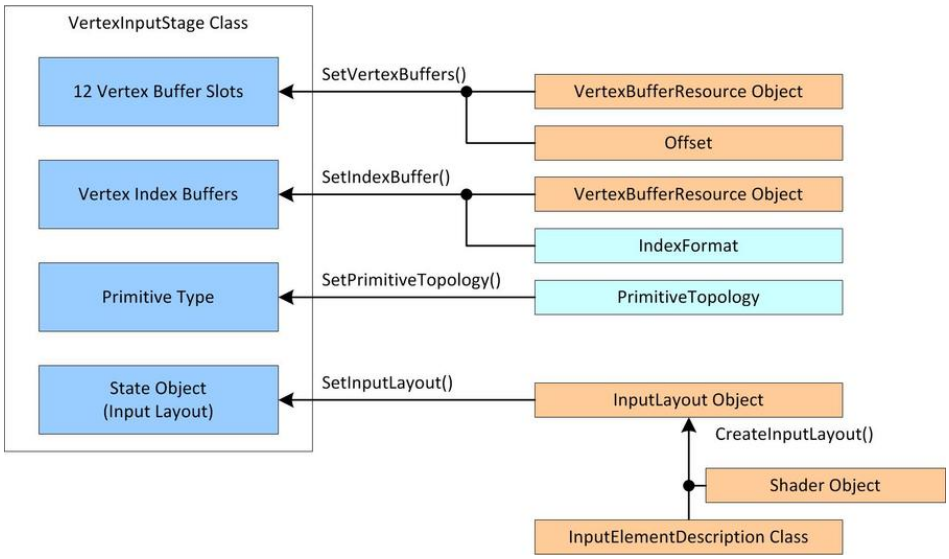
If a ShaderPipeline object has already been applied to the pipeline and you apply a new one that was generated from different ShaderBinary objects, all of the data in the ShaderBinary objects is output to the 3D command buffer during rendering. Because of the associated processing load, we recommend using unmanaged registers instead to minimize the number of 3D commands that are output.

## 4.2. Vertex Input Stage

The vertex input stage handles settings related to vertex data (vertex attributes) input to the pipeline. The nn::gd::VertexInputStage class defines functions that configure these settings.

The following diagram provides an overview of the settings made in the vertex input stage.

**Figure 4-2. Overview of the Vertex Input Stage**




## 4.2.1. Registering Vertex Buffers

Vertex buffers are registered in the vertex input stage's 12 slots. By specifying an array of `VertexBufferResource` objects and an array of starting offsets to vertex data, you can register vertex buffers in multiple consecutive slots by making a single call to the `nn::gd::VertexInputStage::SetVertexBuffers()` function.

**Code 4-10. Registering Vertex Buffers**

```
static nnResult nn::gd::VertexInputStage::SetVertexBuffers(
        u32 startSlot, u32 numBuffers,
        nn::gd::VertexBufferResource** vertexBuffers, u32* offsets);
```

Use the *`startSlot`* parameter to specify the first slot number (0–11) for registering a vertex buffer. Use the *`numBuffers`* parameter to specify the number of elements in the array of `VertexBufferResource` objects given to *`vertexBuffers`*. Use the *`offsets`* parameter to specify the starting offset for the vertex data array. You must register vertex buffers with the same number of vertex data elements defined by the `InputLayout` object (described in 4.2.4. Input Layout).

## 4.2.2. Using Vertex Indices

When you use vertex indices to render primitives, create a `VertexBufferResource` object by specifying an array of vertex indices as resource data, and then call the `nn::gd::VertexInputStage::SetIndexBuffer()` function to register that object to the vertex input stage.

**Code 4-11. Registering Vertex Indices**

```
static nnResult nn::gd::VertexInputStage::SetIndexBuffer(
        nn::gd::VertexBufferResource* indexBuffer,
        nn::gd::VertexInputStage::IndexFormat format, u32 offset);
```

The VertexBufferResource object specified for the *indexBuffer* parameter is registered to the vertex input stage. When you register VertexBufferResource objects interleaved with vertex and other data, specify the starting offset (in bytes) of the vertex indices in the *offset* parameter.

You can specify one of the following vertex index formats for the *format* parameter.

**Table 4-2. Specifying the Vertex Index Format**

| Definition | Description |
|---|---|
| INDEX_FORMAT_UBYTE | Vertex indices are treated as an unsigned byte array. |
| INDEX_FORMAT_USHORT | Vertex indices are treated as an unsigned short array. |

## 4.2.3. Types of Primitives

Use the nn::gd::VertexInputStage::SetPrimitiveTopology() function to set the type of primitives to render.

**Code 4-12. Setting the Type of Primitives to Render**

```
static nnResult nn::gd::VertexInputStage::SetPrimitiveTopology(
        nn::gd::VertexInputStage::PrimitiveTopology primitiveTopology);
```

Specify one of the following types of primitives for the *primitiveTopology* parameter.

**Table 4-3. Types of Primitives**

| Definition | Description |
|---|---|
| PRIMITIVE_TOPOLOGY_TRIANGLELIST | Triangles (GL_TRIANGLES) |
| PRIMITIVE_TOPOLOGY_TRIANGLESTRIP | Triangle strip (GL_TRIANGLE_STRIP) |
| PRIMITIVE_TOPOLOGY_TRIANGLEFAN | Triangle fan (GL_TRIANGLE_FAN) |
| PRIMITIVE_TOPOLOGY_GEOMETRY | Geometry shader (GL_GEOMETRY_PRIMITIVE_DMP) |

## 4.2.4. Input Layout

An input layout (InputLayout object) defines how input variables are connected between a vertex shader and vertex buffer. A vertex buffer comprises either a simple scalar value or several interleaved scalar values. Call the nn::gd::VertexInputStage::CreateInputLayout() function to create an InputLayout object. This function's parameters include an array of descriptor (InputElementDescription) objects for the input elements (vertex data) and a Shader object that was created in the shader stage.

**Code 4-13. Creating, Releasing, and Applying an Input Layout to the Pipeline**

```
static nnResult nn::gd::VertexInputStage::CreateInputLayout(
        nn::gd::InputElementDescription* inputElementDescs, u32 numElements,
        u32* strides, nn::gd::Shader* vertexShader,
        nn::gd::InputLayout** inputLayout);
static nnResult nn::gd::VertexInputStage::ReleaseInputLayout(
        nn::gd::InputLayout* inputLayout);
static nnResult nn::gd::VertexInputStage::SetInputLayout(
```

```
        nn::gd::InputLayout* inputLayout);
```

Specify an array of descriptor (`InputElementDescription`) objects with information on the input elements in *`inputElementDescs`*, and pass the number of descriptor objects in the array to *`numElements`*. For more information about descriptor class settings, see 4.2.4.1. Descriptor Class for InputLayout Objects.

Specify the amount of vertex data used by a single vertex as an array in *`strides`* for each slot number. If all of the input elements are interleaved in a single vertex buffer, this array only has a single element that represents the total size of the input elements. If the input elements are not interleaved, this array has multiple elements for the size of each input element.

Use *`vertexShader`* to specify the `Shader` object for the vertex shader.

You can use the `nn::gd::VertexInputStage::SetInputLayout()` function to apply a created `InputLayout` object to the pipeline.

Use the `nn::gd::VertexInputStage::ReleaseInputLayout()` function to release an `InputLayout` object that is no longer needed. This also deallocates memory regions that were allocated internally when the `CombinerState` object was created.

In the following sample code, an input layout is created from a descriptor class that describes a single vertex buffer with three interleaved input elements. The layout is then applied to the pipeline (vertex input stage).

**Code 4-14. Creating an Input Layout**

```
nn::gd::InputLayout* inputLayout = 0;
nn::gd::InputElementDescription descs[] =
{
    { 0, "aPosition",
        nn::gd::VertexInputStage::STREAM_TYPE_FLOAT, 3, 0 },
    { 0, "aColor",
        nn::gd::VertexInputStage::STREAM_TYPE_FLOAT, 4, sizeof(float) * 3 },
    { 0, "aNormal",
        nn::gd::VertexInputStage::STREAM_TYPE_FLOAT, 3, sizeof(float) * 7 },
};
u32 strides[] = { sizeof(float) * 10 };
nn::gd::VertexInputStage::CreateInputLayout(
        descs, 3, strides, vertexShader, &inputLayout);

nn::gd::VertexInputStage::SetInputLayout(inputLayout);
```

## 4.2.4.1. Descriptor Class for InputLayout Objects

This section introduces the members of the descriptor class (`InputElementDescription`) for `InputLayout` objects and provides setting-related cautions and information.

**Code 4-15. Definition of the InputElementDescription Class**

```
class nn::gd::InputElementDescription
{
    u32 m_StreamSlot;
    const char* m_SemanticName;
    nn::gd::VertexInputStage::ShaderStreamFormatType m_Format;
    u32 m_Count;
    u32 m_AlignedByteOffset;

    static const u32 Append = 0x00080000;
};
```

#### Slot Number (m_StreamSlot)

The slot number (0–11) used by an input element. To use the same slot number for more than one input element, the vertex buffers registered to that slot number must be interleaved. Slot numbers must be registered consecutively in ascending order, starting at 0.

#### Input Variable Name (m_SemanticName)

The name of an input variable defined by the vertex shader.

#### Input Element Data Format (m_Format)

The data format of an input element. Specify one of the values from the following table.

**Table 4-4. Input Element Data Formats**

| Definition | Description |
|---|---|
| `VertexInputStage::STREAM_TYPE_BYTE` | byte (`GL_BYTE`) |
| `VertexInputStage::STREAM_TYPE_UNSIGNED_BYTE` | unsigned byte (`GL_UNSIGNED_BYTE`) |
| `VertexInputStage::STREAM_TYPE_SHORT` | short (`GL_SHORT`) |
| `VertexInputStage::STREAM_TYPE_FLOAT` | float (`GL_FLOAT`) |

#### Input Element Size (m_Count)

The size of (number of components in) an input element. It can be a value between 1 and 4. Note that this is not the size of the input element in bytes.

#### Input Element Offset (m_AlignedByteOffset)

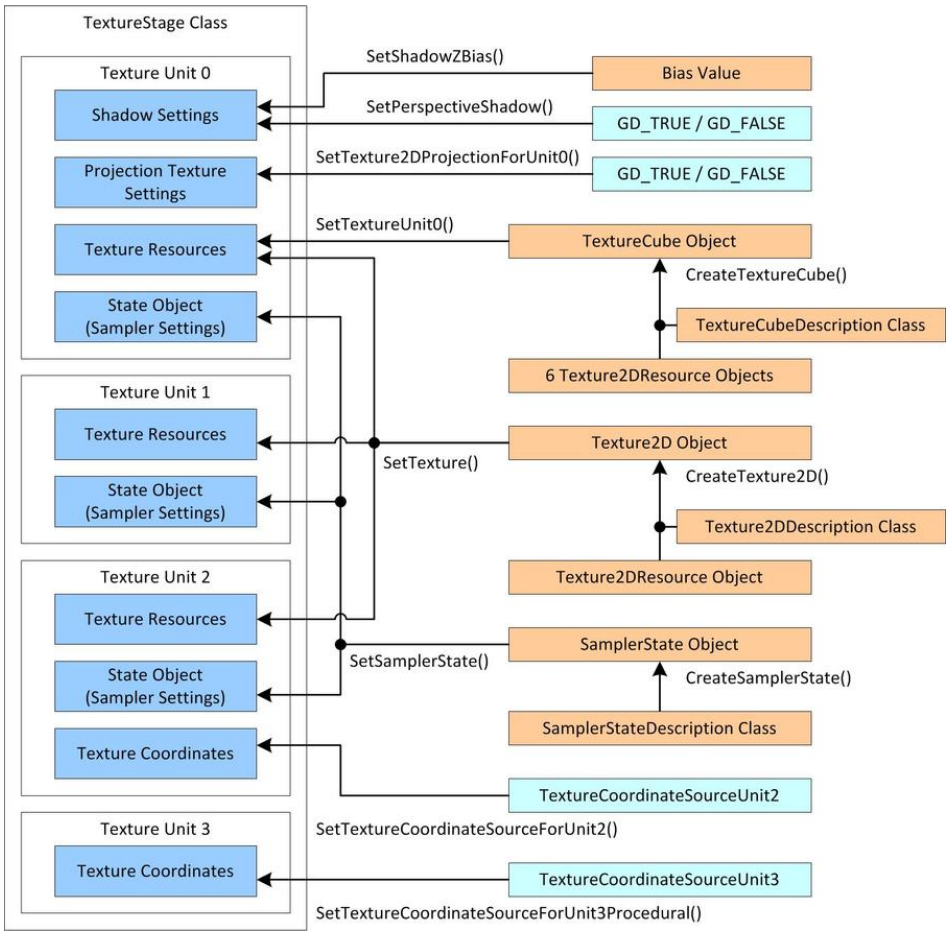
The offset (in bytes) to the starting input element. By specifying `InputElementDescription::Append`, the offset can be calculated automatically, thus simplifying descriptor class settings.

## 4.3. Texture Stage

The texture stage creates texture objects and handles texture unit-related settings. The `nn::gd::TextureStage` class defines functions that configure these settings.

The following diagram provides an overview of the settings made in the texture stage.

**Figure 4-3. Overview of the Texture Stage**




## 4.3.1. 2D Textures

2D textures are handled by `Texture2D` objects in the GD library. A `Texture2D` object is created from a single `Texture2DResource` object and a descriptor class (`Texture2DDescription`). If the `Texture2DResource` object has mipmap data, you can use descriptor class settings to specify which levels of mipmap data to use when you create a `Texture2D` object. For more information about the descriptor class settings, see 4.3.1.1. Descriptor Class for Texture2D Objects.

You can create and release a `Texture2D` object using the following functions.

**Code 4-16. Creating and Releasing a Texture2D Object**

```
static nnResult nn::gd::TextureStage::CreateTexture2D(
        const nn::gd::Texture2DResource* tex2DResource,
        nn::gd::Texture2DDescription* desc,
        nn::gd::Texture2D** texture2D);
static nnResult nn::gd::TextureStage::ReleaseTexture2D(
        nn::gd::Texture2D* texture2D);
```

Only management information is released when a `Texture2D` object is released. `Texture2DResource` objects are not released.

The following sample code creates a `Texture2DResource` object and uses it to create a `Texture2D` object.

**Code 4-17. Creating a Texture2D Object**

```
nn::gd::Texture2DResource* texture2DResource = 0;
nn::gd::Texture2DResourceDescription Text2DResDesc =
    {256, 256, 1, nn::gd::Resource::NATIVE_FORMAT_RGB_888,
    nn::gd::Memory::LAYOUT_BLOCK_8, nn::gd::Memory::FCRAM};
nn::gd::Resource::CreateTexture2DResource(
        &Text2DResDesc, data, GD_FALSE, &texture2DResource);
nn::gd::Texture2D* texture2D = 0;
nn::gd::Texture2DDescription tex2DDesc = {0, 0};
nn::gd::TextureStage::CreateTexture2D(
        texture2DResource, &tex2DDesc, &texture2D);
```

## 4.3.1.1. Descriptor Class for Texture2D Objects

This section introduces the member variables of the descriptor class (`Texture2DDescription`) for `Texture2D` objects and provides setting-related cautions and information.

**Code 4-18. Definition of the Texture2DDescription Class**

```
class nn::gd::Texture2DDescription
{
    s32 m_MinMipLevelIndex;
    s32 m_MaxMipLevelIndex;
};
```

### Index of the Smallest Mipmap Level (m_MinMipLevelIndex)

This is the mipmap level of the `Texture2DResource` object used as the smallest mipmap level of a `Texture2D` object. If you set a value of −1, the largest mipmap level of the `Texture2DResource` object is used.

### Index of the Largest Mipmap Level (m_MaxMipLevelIndex)

This is the mipmap level of the `Texture2DResource` object used as the largest mipmap level of a `Texture2D` object. If you set a value of −1, the largest mipmap level of the `Texture2DResource` object is used.

## 4.3.1.2. Getting Detailed Information About a Texture2D Object

You can use the `nn::gd::TextureStage::GetTexture2DProperties()` function to get detailed information (using the `Texture2DProperties` class).

**Code 4-19. Getting Detailed Information on a Texture2D Object**

```
static nnResult nn::gd::TextureStage::GetTexture2DProperties(
        const nn::gd::Texture2D* texture2D,
        nn::gd::Texture2DProperties* properties);
```

Although the width, height, mipmap levels, and starting address of resource data may change with the mipmap levels specified when an object was created, they are generally the same as the detailed information for the `Texture2DResource` object.

**Code 4-20. Definition of the Texture2DProperties Class**

```
class nn::gd::Texture2DProperties
{
    u32 m_Width;
    u32 m_Height;
    u32 m_CountMipLevels;
    u32 m_PixelSize;
    nn::gd::Resource::NativeFormat m_Format;
    nn::gd::Memory::MemoryLayout m_MemLayout;
    nn::gd::Memory::MemoryLocation m_MemLocation;
    u8* m_MemAddr;
    nn::gd::MipmapResourceInfo GetMipmapAddress(u32 mipmapLevel);
};
```

## 4.3.2. Cube Map Textures

Cube map textures are handled by `TextureCube` objects in the GD library. A `TextureCube` object is created using six `Texture2DResource` objects. As with `Texture2D` objects, you can specify which mipmap levels to use. Each face (`Texture2DResource` object) of the cube map texture must have the same format, mipmap levels, resolution (which depends on the mipmap level), and memory placement.

You can create and release a `TextureCube` object using the following functions.

**Code 4-21. Creating and Releasing a TextureCube Object**

```
static nnResult nn::gd::TextureStage::CreateTextureCube(
        nn::gd::Texture2DResource** tex2DResources,
        nn::gd::TextureCubeDescription* desc,
        nn::gd::TextureCube** textureCube);
static nnResult nn::gd::TextureStage::ReleaseTextureCube(
        nn::gd::TextureCube* textureCube);
```

Specify an array of six `Texture2DResource` objects, one for each face of the cube map texture, in the *`tex2DResources`* parameter. These textures are used for the +X, -X, +Y, -Y, +Z, and -Z faces, respectively, from the beginning of the array. Cube map texture faces are specified in the same order as the elements in descriptor class arrays.

Only management information is released when a `TextureCube` object is released. `Texture2DResource` objects are not released.

### 4.3.2.1. Descriptor Class for TextureCube Objects

This section introduces the members of the descriptor class (`TextureCubeDescription`) for `TextureCube` objects and provides setting-related cautions and information.

**Code 4-22. Definition of the TextureCubeDescription Class**

```
class nn::gd::TextureCubeDescription
{
    s32 m_MinMipLevelIndex[6];
    s32 m_MaxMipLevelIndex[6];
    TextureCubeDescription();
    TextureCubeDescription(int minMipLevelIndex, int maxMipLevelIndex);
};
```

This class differs from `Texture2DDescription` in that it specifies mipmap levels for six faces and has a constructor to simplify the process of doing so.

#### Index of the Smallest Mipmap Level (m_MinMipLevelIndex)

This is the mipmap level of the `Texture2DResource` object used as the smallest mipmap level of a `TextureCube` object. If you set a value of –1, the largest mipmap level of the `Texture2DResource` object is used.

The values in the array correspond to the `Texture2DResource` objects for the +X, -X, +Y, -Y, +Z, and -Z faces, in that order.

#### Index of the Largest Mipmap Level (m_MaxMipLevelIndex)

This is the mipmap level of the `Texture2DResource` object used as the largest mipmap level of a `TextureCube` object. If you set a value of –1, the largest mipmap level of the `Texture2DResource` object is used.

The values in the array correspond to the `Texture2DResource` objects for the +X, -X, +Y, -Y, +Z, and -Z faces, in that order.

## 4.3.2.2. Getting Detailed Information About a TextureCube Object

You can use the `nn::gd::TextureStage::GetTextureCubeProperties()` function to get detailed information (using the `TextureCubeProperties` class) about a `TextureCube` object.

**Code 4-23. Getting Detailed Information About a TextureCube Object**

```
static nnResult nn::gd::TextureStage::GetTextureCubeProperties(
        const nn::gd::TextureCube* TextureCube,
        nn::gd::TextureCubeProperties* properties);
```

This class is the same as `Texture2DProperties` with the following exceptions: it maintains the starting address of six faces of resource data and accepts a cube map face in the *`faceIndex`* parameter, to get address information for mipmap data.

**Code 4-24. Definition of the TextureCubeProperties Class**

```
class nn::gd::TextureCubeProperties
{
    u32 m_Width;
    u32 m_Height;
    u32 m_CountMipLevels;
    u32 m_PixelSize;
    nn::gd::Resource::NativeFormat m_Format;
    nn::gd::Memory::MemoryLayout m_MemLayout;
    nn::gd::Memory::MemoryLocation m_MemLocation;
    u8* m_MemAddr[6];

    nn::gd::MipmapResourceInfo GetMipmapAddress(u32 faceIndex, u32
mipmapLevel);
};
```

# 4.3.3. Texture Units

The GD library provides four logical texture units to match the 3DS hardware.

**Table 4-5. Types of Textures That Can Be Set for Each Texture Unit**

| Definition | Types of Textures That Can Be Set |
|---|---|
| `TextureStage::TEXTURE_UNIT_0`<br>(Texture unit 0) | 2D textures (including gas and shadows)<br>Cube map textures (including shadows)<br>Projection textures |
| `TextureStage::TEXTURE_UNIT_1`<br>(Texture unit 1) | 2D textures |
| `TextureStage::TEXTURE_UNIT_2`<br>(Texture unit 2) | 2D textures |
| `TextureStage::TEXTURE_UNIT_3_PROCEDURAL`<br>(Texture unit 3) | Procedural textures |

**Note:** The procedural texture stage handles all `TextureStage::TEXTURE_UNIT_3_PROCEDURAL` settings except for texture coordinates.

### 2D textures

Use the `nn::gd::TextureStage::SetTexture()` function to set a 2D texture. You can set a 2D texture in three texture units: 0, 1, and 2.

**Code 4-25. Setting a 2D Texture in a Texture Unit**

```
static nnResult nn::gd::TextureStage::SetTexture(
        nn::gd::TextureStage::TextureUnitId textureUnitId,
        nn::gd::Texture2D& texture2D);
```

### Other (Non-Procedural) Textures

All other textures (except for procedural textures) can only be set in texture unit 0.

**Code 4-26. Setting Other Textures**

```
static nnResult nn::gd::TextureStage::SetTextureUnit0(
        nn::gd::TextureCube& textureCube);
static void nn::gd::TextureStage::SetTexture2DProjectionForUnit0(gdBool value);
static void nn::gd::TextureStage::SetPerspectiveShadow(gdBool v);
static void nn::gd::TextureStage::SetShadowZBias(f32 zBias);
```

Use the `nn::gd::TextureStage::SetTextureUnit0()` function to set a cube map texture.

By calling the `nn::gd::TextureStage::SetTexture2DProjectionForUnit0()` function and passing GD_TRUE as an argument, you can treat the 2D texture set in texture unit 0 as a projection texture.

If you apply perspective projection to create texture coordinates when you access a shadow texture or shadow cube map texture, pass GD_TRUE as an argument to

`nn::gd::TextureStage::SetPerspectiveShadow`. You can also use `nn::gd::TextureStage::SetShadowZBias` to set a bias value that will be subtracted from the distance to the light source when a shadow texture is generated.

#### 4.3.3.1. Texture Coordinates

The following functions set the texture coordinates input to texture units 2 and 3.

**Code 4-27. Setting Texture Coordinates**

```
// For TextureUnit2
static void nn::gd::TextureStage::SetTextureCoordinateSourceForUnit2(
        nn::gd::TextureStage::TextureCoordinateSourceUnit2 u);
// For TextureUnit3
static void nn::gd::TextureStage::SetTextureCoordinateSourceForUnit3Procedural(
        nn::gd::TextureStage::TextureCoordinateSourceUnit3Procedural u);
```

You can specify the following arguments to these functions.

**Table 4-6. Texture Coordinates That Can Be Set for Texture Unit 2**

| Definition | Input Texture Coordinates |
|---|---|
| UNIT2_TEXTURE_COORDINATE_FROM_UNIT_2 | Texture coordinate 2 (default). |
| UNIT2_TEXTURE_COORDINATE_FROM_UNIT_1 | Texture coordinate 1 |

**Table 4-7. Texture Coordinates That Can Be Set for Texture Unit 3**

| Definition | Input Texture Coordinates |
|---|---|
| UNIT3_PROCEDURAL_TEXTURE_COORDINATE_FROM_UNIT_0 | Texture coordinate 0 (default). |
| UNIT3_PROCEDURAL_TEXTURE_COORDINATE_FROM_UNIT_1 | Texture coordinate 1 |
| UNIT3_PROCEDURAL_TEXTURE_COORDINATE_FROM_UNIT_2 | Texture coordinate 2 |

### 4.3.4. Sampler Settings

To configure the texture sampling method (sampler settings), create a state (`SamplerState`) object and set it in a texture unit.

**Code 4-28. Creating, Releasing, and Specifying Sampler Settings in a Texture Unit**

```
static nnResult nn::gd::TextureStage::CreateSamplerState(
        const nn::gd::SamplerStateDescription* desc,
        nn::gd::SamplerState** sampler);
static nnResult nn::gd::TextureStage::ReleaseSamplerState(
        nn::gd::SamplerState* sampler);
static nnResult nn::gd::TextureStage::SetSamplerState(
        nn::gd::TextureStage::TextureUnitId textureUnitId,
        nn::gd::SamplerState* sampler);
```

Configure sampler settings in a descriptor class (`SamplerStateDescription`) in advance, specified in the *desc* parameter. For more information about descriptor class settings, see 4.3.4.1.

Descriptor Class for SamplerState Objects.

You can use the `nn::gd::TextureStage::SetSamplerState()` function to set a created `SamplerState` object in a texture unit. You can configure sampler settings in three texture units: 0, 1, and 2.

Use the `nn::gd::TextureStage::ReleaseSamplerState()` function to release a `SamplerState` object that is no longer needed. This also deallocates memory regions that were allocated internally when the `CombinerState` object was created.

The following sample code creates a `SamplerState` object with the default settings, changes the minification and magnification filters, and then sets that object in texture unit 0.

**Code 4-29. Sampler Settings**

```
nn::gd::SamplerState* sampler = 0;
nn::gd::SamplerStateDescription samplerdesc;
samplerdesc.ToDefault();
samplerdesc.m_MinFilter = nn::gd::TextureStage::SAMPLER_MIN_FILTER_LINEAR;
samplerdesc.m_MagFilter = nn::gd::TextureStage::SAMPLER_MAG_FILTER_LINEAR;
nn::gd::TextureStage::CreateSamplerState(&samplerdesc, &sampler);
nn::gd::TextureStage::SetSamplerState(
        nn::gd::TextureStage::TEXTURE_UNIT_0, sampler);
```

## 4.3.4.1. Descriptor Class for SamplerState Objects

This section introduces the members of the descriptor (`SamplerStateDescription`) class for `SamplerState` objects and provides setting-related cautions and information.

**Code 4-30. Definition of the SamplerStateDescription Class**

```
class nn::gd::SamplerStateDescription
{
    nn::gd::TextureStage::SamplerMinFilter m_MinFilter;
    nn::gd::TextureStage::SamplerMagFilter m_MagFilter;
    nn::gd::TextureStage::SamplerWrapMode m_WrapS;
    nn::gd::TextureStage::SamplerWrapMode m_WrapT;
    u8 m_BorderColor[4];
    u32 m_LodBias;
    u32 m_MinLod;
    u32 m_MaxLod;

    void SetMinFilter(nn::gd::TextureStage::SamplerMinFilter filter);
    void SetMagFilter(nn::gd::TextureStage::SamplerMagFilter filter);
    void SetWrapS(nn::gd::TextureStage::SamplerWrapMode wrap);
    void SetWrapT(nn::gd::TextureStage::SamplerWrapMode wrap);
    void SetBorderColor(u8 colorR, u8 colorG, u8 colorB, u8 colorA);
    void SetBorderColor(f32 colorR, f32 colorG, f32 colorB, f32 colorA);
    void SetLodBias(f32 biasValue);
    void SetMinLod(u32 lodValue);
    void SetMaxLod(u32 lodValue);

    SamplerStateDescription();
    void ToDefault();
    void SetShadow();
    void SetShadowCube();
    void SetGas();
};
```

### Minification Filter (m_MinFilter)

This filter is applied during texture minification. You can set this value using the

SetMinFilter() function.

### Magnification Filter (m_MagFilter)

This filter is applied during texture magnification. You can set this value using the SetMagFilter() function.

### Wrapping Mode in the S Direction (m_WrapS)

This wrapping mode is applied to repeating textures in the S direction. You can set this value using the SetWrapS() function.

### Wrapping Mode in the T Direction (m_WrapT)

This wrapping mode is applied to repeating textures in the T direction. You can set this value using the SetWrapT() function.

### Border Color (m_BorderColor)

This border color is used when TextureStage::SAMPLER_WRAP_CLAMP_TO_BORDER is specified as the wrapping mode. You can call the SetBorderColor() function to set each RGBA component to a value between 0 and 255 or between 0.0 and 1.0.

### LOD Bias (m_LodBias)

This bias value is used to determine the level of detail (LOD). Because you must set this value using the same format as when setting a GPU register, do so by passing a value between –16.0 and 16.0 to the SetLoadBias() function.

The bias value is a 13-bit signed fixed-point decimal number with 8 fractional bits. To change member variables directly, use the Utils::Float32ToFix13Fraction8 utility function.

### Minimum LOD (m_MinLod)

This is the smallest mipmap level used for LOD. If you use LOD, set the smallest mipmap level to a value of 0 or greater. (Specify 0 if you do not use LOD.) You can set this value using the SetMinLod() function.

### Maximum LOD (m_MaxLod)

This is the largest mipmap level used for LOD. Specify a value of 0 when you do not use LOD and a value that is one less than the maximum number of mipmap levels in the texture when you do use LOD. You can set this value using the SetMaxLod() function.

Specify UINT_MAX or another large value when the same sampler settings are used by textures that have different numbers of mipmap levels. This value will be one less than the number of mipmap levels in the texture when it is rendered. It is determined from the default settings and source.

### Default Settings (ToDefault)

You can call the `ToDefault()` function to change all members to their default values. This is also done in the constructor.

**Table 4-8. Default Sampler Settings**

| Member | Setting Value |
|---|---|
| `m_MinFilter` | `TextureStage::SAMPLER_MIN_FILTER_NEAREST` |
| `m_MagFilter` | `TextureStage::SAMPLER_MAG_FILTER_NEAREST` |
| `m_WrapS` | `TextureStage::SAMPLER_WRAP_REPEAT` |
| `m_WrapT` | `TextureStage::SAMPLER_WRAP_REPEAT` |
| `m_BorderColor` | (0, 0, 0, 0) |
| `m_LodBias` | 0.0 |
| `m_MinLod` | 0 |
| `m_MaxLod` | 0xFFFFFFFF |

### Shadow Texture Settings (SetShadow)

You can call the `SetShadow()` function to change all members to their values for shadow textures.

**Table 4-9. Sampler Settings for Shadow Textures**

| Member | Setting Value |
|---|---|
| `m_MinFilter` | `TextureStage::SAMPLER_MIN_FILTER_LINEAR` |
| `m_MagFilter` | `TextureStage::SAMPLER_MAG_FILTER_LINEAR` |
| `m_WrapS` | `TextureStage::SAMPLER_WRAP_CLAMP_TO_BORDER` |
| `m_WrapT` | `TextureStage::SAMPLER_WRAP_CLAMP_TO_BORDER` |
| `m_BorderColor` | (0, 0, 0, 0) |
| `m_LodBias` | 0.0 |
| `m_MinLod` | 0 |
| `m_MaxLod` | 0xFFFFFFFF |

**Warning:** Do not change the filter, wrapping mode, and LOD settings.

### Shadow Cube Texture Settings (SetShadowCube)

You can call the `SetShadowCube()` function to change all members to their values for shadow cube map textures.

**Table 4-10. Sampler Settings for Shadow Cube Map Textures**

| Member | Setting Value |
|---|---|
| `m_MinFilter` | `TextureStage::SAMPLER_MIN_FILTER_LINEAR` |
| `m_MagFilter` | `TextureStage::SAMPLER_MIN_FILTER_LINEAR` |
| `m_WrapS` | `TextureStage::SAMPLER_WRAP_CLAMP_TO_EDGE` |
| `m_WrapT` | `TextureStage::SAMPLER_WRAP_CLAMP_TO_EDGE` |

| | |
|---|---|
| m_BorderColor | (0, 0, 0, 0) |
| m_LodBias | 0.0 |
| m_MinLod | 0 |
| m_MaxLod | 0xFFFFFFFF |

**Warning:** Do not change the filter, wrapping mode, and LOD settings.

### Gas Texture Settings (SetGas)

You can call the `SetGas()` function to change all members to their values for gas textures.

**Table 4-11. Sampler Settings for Gas Textures**

| Member | Setting Value |
|---|---|
| m_MinFilter | TextureStage::SAMPLER_MIN_FILTER_NEAREST |
| m_MagFilter | TextureStage::SAMPLER_MAG_FILTER_NEAREST |
| m_WrapS | TextureStage::SAMPLER_WRAP_CLAMP_TO_EDGE |
| m_WrapT | TextureStage::SAMPLER_WRAP_CLAMP_TO_EDGE |
| m_BorderColor | (0, 0, 0, 0) |
| m_LodBias | 0.0 |
| m_MinLod | 0 |
| m_MaxLod | 0xFFFFFFFF |

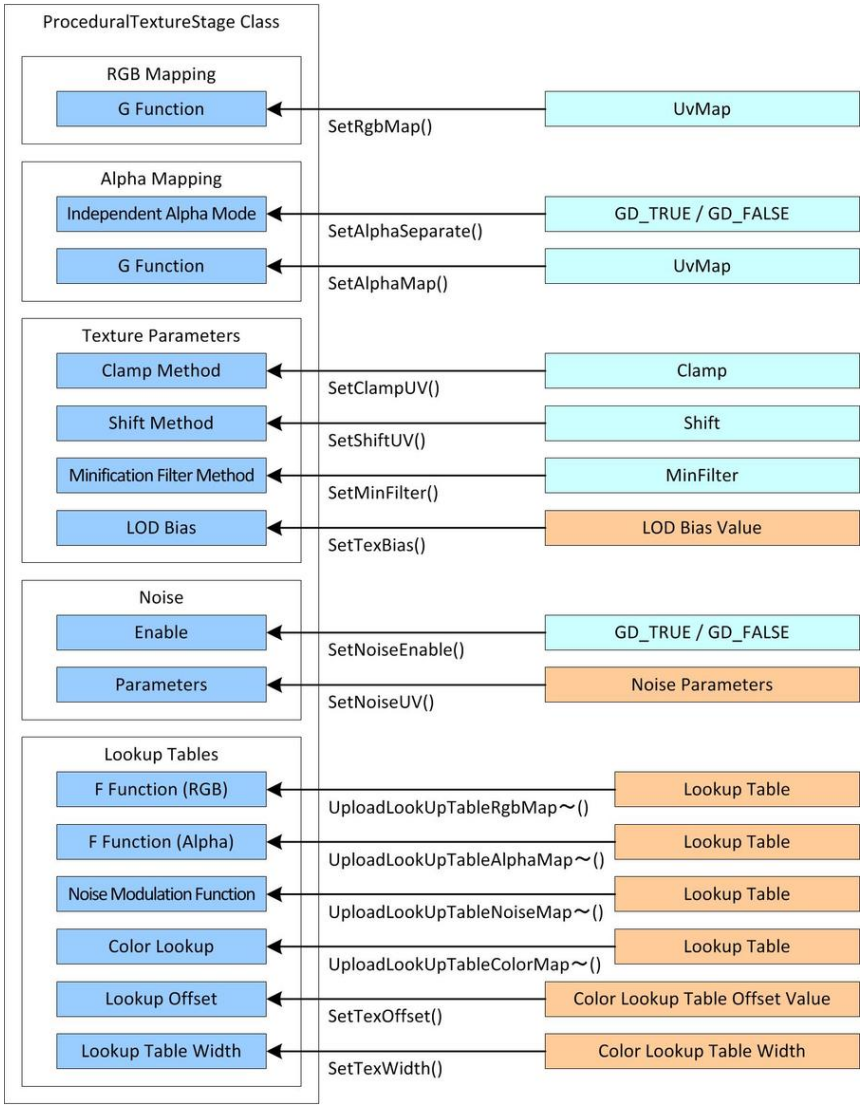
**Warning:** Do not change the filter, wrapping mode, and LOD settings.

## 4.4. Procedural Texture Stage

The procedural texture stage handles settings related to procedural textures. The `nn::gd::ProceduralTextureStage` class defines functions that configure these settings.

The following figure gives an overview of the settings made in the procedural texture stage.

**Figure 4-4. Overview of the Procedural Texture Stage**




## 4.4.1. Setting Procedural Texture Parameters

Functions have been provided for setting each of the procedural texture parameters.

**Code 4-31. Functions for Setting Procedural Texture Parameters**

```
class nn::gd::ProceduralTextureStage
{
    static void SetRgbMap(UvMap rgbMap);
    static void SetAlphaSeparate(gdBool alphaSeparate);
    static void SetAlphaMap(UvMap alphaMap);
    static void SetClampUV(Clamp u, Clamp v);
    static void SetShiftUV(Shift u, Shift v);
    static void SetMinFilter(MinFilter minFilter);
    static void SetTexBias(f32 texBias);
    static void SetNoiseEnable(gdBool noiseEnable);
    static void SetNoiseUV(f32 noiseU[3], f32 noiseV[3]);
    static void SetTexWidth(u8 texWidth);
    static void SetTexOffset(u8 texOffset);
};
```

The following table shows which parameters are set by each function. For more information about how to specify arguments and what effect these parameters have, see the *CTR-SDK API Reference*

and other documentation.

**Table 4-12. Parameters Set by Functions in the Procedural Texture Stage**

| Function | Parameters |
|---|---|
| SetRgbMap | The G function for RGB mapping. Default value: UV_MAP_U |
| SetAlphaSeparate | Whether to enable independent alpha mode. Default value: GD_FALSE |
| SetAlphaMap | The G function for alpha mapping. Default value: UV_MAP_U |
| SetClampUV | The clamp mode. Default value: CLAMP_TO_ZERO |
| SetShiftUV | The shift mode. Default value: SHIFT_NONE |
| SetMinFilter | The minification filter mode. Default value: MIN_FILTER_LINEAR |
| SetTexBias | The LOD bias. Default value: 0.5 |
| SetNoiseEnable | Whether to enable noise. Default value: GD_FALSE |
| SetNoiseUV | Noise parameters. Default values: F=0.0, P=0.0, and A=0.0 for both U and V |
| SetTexWidth | Color lookup table width. Default value: 0 |
| SetTexOffset | Color lookup table offset. Default value: 0 |

**Note:** The SetTexBias and SetNoiseUV() functions convert the values passed as arguments into the format used when setting GPU registers.

## 4.4.2. Loading Lookup Tables Used by the Procedural Texture Stage

There are two ways to load F functions (for RGB mapping and alpha mapping), noise modulation data, color lookup tables, and other lookup tables used by procedural textures: The first is to use functions (with the Float suffix) that take arrays of floating-point numbers. The second is to use functions (with the Native suffix) that take arrays of data to be written unchanged to the GPU registers. Because these are immediate functions, data is written to the 3D command buffer when they are run.

**Code 4-32. Functions for Loading Lookup Tables Used by the Procedural Texture Stage**

```
class nn::gd::ProceduralTextureStage
{
    static nnResult UploadLookUpTableRgbMapFloat(
            u32 index, f32* Map, f32* MapDelta, u32 lutSize);
    static nnResult UploadLookUpTableRgbMapNative(
            u32 index, u32* Map, u32 lutSize);
    static nnResult UploadLookUpTableAlphaMapFloat(
            u32 index, f32* Map, f32* MapDelta, u32 lutSize);
    static nnResult UploadLookUpTableAlphaMapNative(
            u32 index, u32* Map, u32 lutSize);
    static nnResult UploadLookUpTableNoiseMapFloat(
            u32 index, f32* Map, f32* MapDelta, u32 lutSize);
    static nnResult UploadLookUpTableNoiseMapNative(
            u32 index, u32* Map, u32 lutSize);
    static nnResult UploadLookUpTableColorMapFloat(
            u32 index, f32** Map, f32** MapDelta, u32 lutSize);
    static nnResult UploadLookUpTableColorMapNative(
            u32 index, u32* Map, u32* MapDelta, u32 lutSize);
};
```

You can use *index* and *lutSize* values to partially overwrite a lookup table. However, an error occurs if the total exceeds the maximum number of elements in the following table.

To load a lookup table with arrays of floating-point numbers, you must provide one array for the data (the *Map* parameter) and one for the deltas (the *MapDelta* parameter). However, you need eight arrays to load a color lookup table: two for each of the color's four components.

If you load any lookup table (except for a color lookup table) with an array that has been converted into native format, you only need to provide one array because both the data and the delta have been combined into a single element.

The following table shows the format of and maximum number of elements in the arrays used by each function to load a lookup table.

**Table 4-13. Functions for Loading Lookup Tables Used by the Procedural Texture Stage**

| Function | Lookup Table | Format | Maximum Number of Elements |
|---|---|---|---|
| UploadLookUpTableRgbMapFloat | RGB mapping | Floating-point numbers | 128 |
| UploadLookUpTableRgbMapNative | RGB mapping | Native | 128 |
| UploadLookUpTableAlphaMapFloat | Alpha mapping | Floating-point numbers | 128 |
| UploadLookUpTableAlphaMapNative | Alpha mapping | Native | 128 |
| UploadLookUpTableNoiseMapFloat | Noise modulation table | Floating-point numbers | 128 |
| UploadLookUpTableNoiseMapNative | Noise modulation table | Native | 128 |
| UploadLookUpTableColorMapFloat | Color lookup table | Floating-point numbers | 256 |
| UploadLookUpTableColorMapNative | Color lookup table | Native | 256 |

## 4.4.2.1. Helper Functions for Loading Lookup Tables

Helper functions have been provided for converting arrays of floating-point numbers into native format. Functions that load lookup tables from arrays of floating-point numbers internally convert the arrays into native format. This is expensive and not suited for runtime use. We recommend that you use some other strategy, such as providing data that has already been converted into native format or using the helper functions to convert the data when the lookup table is loaded for the first time.

**Code 4-33. Helper Functions for Loading Lookup Tables**

```
class nn::gd::ProceduralTextureStage::Helper
{
    static nnResult ConvertLookUpTableDataFloatToNative(
            f32* valueData, f32* deltaData, u32 lutSize, u32* destination);
    static nnResult ConvertColorLookUpTableDataFloatToNative(
            f32** refArray, f32** deltaArray, u32 lutSize,
            u32* destRef, u32* destDelta);
};
```

For all lookup tables except for color lookup tables, a single array of data (*destination*) is generated by combining the data and deltas. The original data is converted into 12-bit unsigned fixed-point numbers with 12 fractional bits in bits [11:0] of the generated data. The original deltas are converted into 12-bit signed fixed-point numbers with 11 fractional bits (and negative numbers

expressed in two's complement) in bits [23:12] of the generated data.

For color lookup tables, a combination of data and deltas for each RGBA component (a total of eight arrays) is converted into a combination of a single color and delta value (the *destRef* and *destDelta* parameters). Although they are converted through different methods, every color and delta value that is generated has eight bits each for the alpha, blue, green, and red components, arranged respectively from the most- to the least-significant bit. Color values are converted from 8-bit signed fixed-point numbers with 7 fractional bits, between `0.0` and `1.0`, into unsigned 8-bit integers between `0` and 255. Delta values between `–1.0` and `1.0` are converted into 8-bit signed (two's complement) fixed-point numbers with 7 fractional bits.

**Note:** For more information about how these values are converted, see the *3DS Programming Manual: Advanced Graphics*.

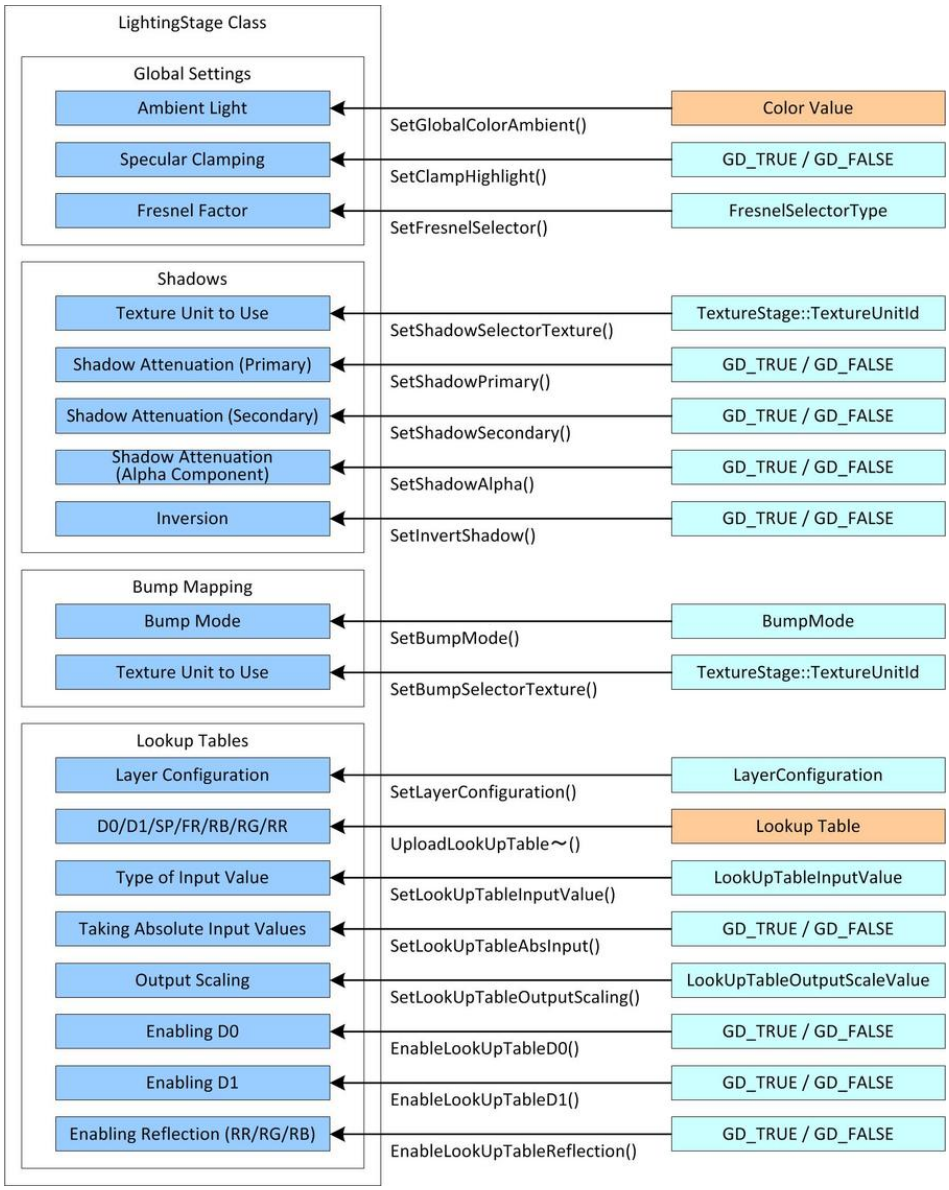
The GD library provides utility functions for these conversions.

## 4.5. Lighting Stage

The lighting stage configures the global lighting environment and individual lights. The `nn::gd::LightingStage` class defines functions that configure these settings. Applications do not need to enable or disable the lighting environment. The library manages lighting settings. The lighting environment is only enabled when lighting results are required by combiner settings.

The following diagram provides an overview of the settings made in the lighting stage. (This does not include individual lights).

**Figure 4-5. Overview of the Lighting Stage (Except for Individual Lights)**




## 4.5.1. Configuring the Global Lighting Environment

Functions have been provided for setting each of the global lighting environment parameters.

**Code 4-34. Functions for Setting Global Lighting Parameters**

```
class nn::gd::LightingStage
{
    static void SetGlobalColorAmbient(u8 R, u8 G, u8 B);
    static void SetClampHightlight(gdBool value);
    static void SetFresnelSelector(FresnelSelectorType fresnelSelectorType);

    static void SetShadowSelectorTexture(
        nn::gd::TextureStage::TextureUnitId textureUnit);
    static void SetShadowPrimary(gdBool value);
    static void SetShadowSecondary(gdBool value);
    static void SetShadowAlpha(gdBool value);
    static void SetInvertShadow(gdBool value);

    static void SetBumpMode(BumpMode bumpMode, gdBool bumpRenorm);
    static void SetBumpSelectorTexture(
        nn::gd::TextureStage::TextureUnitId textureUnit);
```

```
};
```

The following table shows which parameters are set by each function. For more information about how to specify arguments and what effect these parameters have, see the *CTR-SDK API Reference* and other documentation.

**Table 4-14. Parameters Configured by Functions in the Lighting Stage**

| Function | Parameters |
|---|---|
| SetGlobalColorAmbient | The light source color of global ambient light. Default value: (10, 10, 10) |
| SetClampHightlight | Specular light clamping. Default value: GD_TRUE |
| SetFresnelSelector | Fresnel factor. Default value: FRESNEL_SELECTOR_TYPE_NO_FRESNEL |
| SetShadowSelectorTexture | The texture unit used by shadows. Default value: TEXTURE_UNIT_0 |
| SetShadowPrimary | Effect of shadows on the primary color. Default value: GD_FALSE |
| SetShadowSecondary | Effect of shadows on the secondary color. Default value: GD_FALSE |
| SetShadowAlpha | Effect of shadows on the alpha component. Default value: GD_FALSE |
| SetInvertShadow | Shadow term inversion. Default value: GD_FALSE |
| SetBumpMode | Bump-map perturbation mode and whether to regenerate the third component of normal vectors. Default values: BUMPMODE_NOT_USED, GD_FALSE |
| SetBumpSelectorTexture | The texture unit used for bump mapping. Default value: TEXTURE_UNIT_0 |

**Note:** SetGlobalColorAmbient is an immediate function.

## 4.5.2. Setting the Layer Configuration and Lookup Tables

Functions have been provided to set the following: the layer configuration (this determines which lookup tables are used for lighting), lookup table input values, scaling of lookup table output values, distribution factors, and reflections.

**Code 4-35. Functions for Setting Lookup Table-Related Parameters**

```
class nn::gd::LightingStage
{
    static void SetLayerConfiguration(LayerConfiguration layerConfiguration);
    static void SetLookUpTableInputValue(
        LookUpTableId lutId, LookUpTableInputValue lutInputValue);
    static void SetLookUpTableAbsInput(LookUpTableId lutId, gdBool value);
    static void SetLookUpTableOutputScaling(
        LookUpTableId lutId, LookUpTableOutputScaleValue outputScalingValue);
    static void EnableLookUpTableD0(gdBool value);
    static void EnableLookUpTableD1(gdBool value);
    static void EnableLookUpTableReflection(gdBool value);
};
```

The following table shows which parameters are set by each function. For more information about how to specify arguments and what effect these parameters have, see the *CTR-SDK API Reference* and other documentation.

**Table 4-15. Lookup Table-Related Parameters Set by Functions in the Lighting Stage**

| Function | Parameters |
|---|---|
| `SetLayerConfiguration` | The layer configuration. Default value: `LAYER_CONFIGURATION_0` |
| `SetLookUpTableInputValue` | Lookup table input value. Default value: `INPUT_VALUE_NH` for all lookup tables |
| `SetLookUpTableAbsInput` | Whether to take the absolute value of lookup table input value. Default value: `GD_FALSE` for all lookup tables |
| `SetLookUpTableOutputScaling` | How to scale lookup table output values. Default value: `OUTPUT_SCALE_VALUE_1` for all lookup tables |
| `EnableLookUpTableD0` | Whether to apply lookup tables to distribution factor 0. Default value: `GD_FALSE` |
| `EnableLookUpTableD1` | Whether to apply lookup tables to distribution factor 1. Default value: `GD_FALSE` |
| `EnableLookUpTableReflection` | Whether to apply lookup tables to reflections. Default value: `GD_FALSE` |

## 4.5.3. Loading Lookup Tables Used by the Lighting Stage

There are two functions for loading lookup tables used by lighting: `UploadLookUpTableFloat`, which takes arrays of floating-point numbers, and `UploadLookUpTableNative`, which takes arrays of data to be written unchanged to the GPU registers. Because these are immediate functions, data is written to the 3D command buffer when they are run.

**Code 4-36. Functions for Loading Lookup Tables Used by the Lighting Stage**

```
class nn::gd::LightingStage
{
    static nnResult UploadLookUpTableFloat(
        LookUpTableUploadId lutID, u32 lutStartIndex,
        const f32* valueData, const f32* deltaData, u32 dataCount);
    static nnResult UploadLookUpTableNative(
        LookUpTableUploadId lutID, u32 lutStartIndex,
        const u32* data, u32 dataCount);

    class Helper
    {
        static nnResult ConvertLookUpTableDataFloatToNative(
            const f32* valueData, const f32* deltaData, u32 dataCount,
            u32* __restrict destination);
    };
};
```

Lighting uses lookup tables with 256 elements. You can use *`lutStartIndex`* and *`dataCount`* values to partially overwrite a lookup table, but an error will occur if the total exceeds 256.

A lookup table's structure differs depending upon whether its input values are converted into absolute values. Note that the input values corresponding to the 128th and 129th elements of a lookup table will be discontinuous (they will jump from `0.9922` to `−1.0`) if the lookup table input values are not converted into absolute values (if they are between `−1.0` and `1.0`).

To load a lookup table with arrays of floating-point numbers, you need to provide one array for the data (the *`valueData`* parameter) and one array for the deltas (the *`deltaData`* parameter).

If you load any lookup table with an array that has been converted into native format, you only need to provide one array because both the data and the delta have been combined into a single element.

Helper functions have been provided for converting arrays of floating-point numbers into native format. These helper functions generate a single array of data (the *destination* parameter) by combining the data and deltas. The original data is converted into 12-bit unsigned fixed-point numbers with 12 fractional bits. The original deltas are converted into 12-bit signed fixed-point numbers with 11 fractional bits (which are absolute values). These are stored in bits [11:0] and [23:12] of the generated data, respectively.

**Note:** For more information about how these values are converted, see the *3DS Programming Manual: Advanced Graphics*.

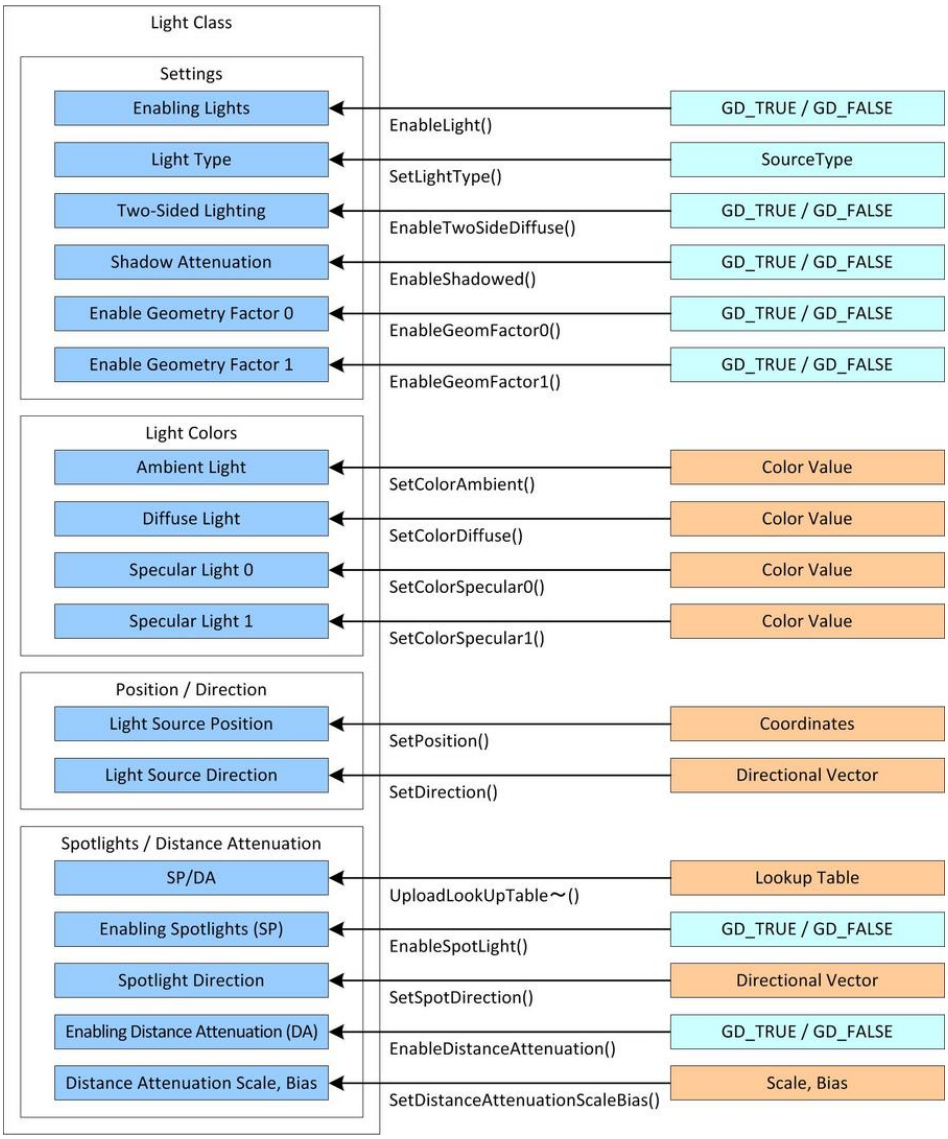
The GD library provides utility functions for these conversions.

## 4.5.4. Lights

The 3DS hardware supports lighting with 8 lights. The GD library represents individual lights as separate instances of the `nn::gd::Light` class. The `nn::gd::LightingStage` class contains eight `Light` objects that are created when the library is initialized. As a result, applications do not need to create `Light` objects. Access them using the members defined by the lighting stage. For example, you could call `nn::gd::LightingStage::light[0].EnableLight(GD_TRUE)`.

The following diagram provides an overview of the settings handled by `Light` objects.

**Figure 4-6. Overview of Light Objects**




## 4.5.4.1. Light Settings

Functions have been provided for setting each of the light parameters.

**Code 4-37. Functions for Setting Light Parameters**

```
class nn::gd::Light
{
    void EnableLight(gdBool value);
    void SetLightType(SourceType sourceType);
    void EnableTwoSideDiffuse(gdBool value);
    void EnableShadowed(gdBool value);
    void EnableGeomFactor0(gdBool value);
    void EnableGeomFactor1(gdBool value);

    void SetColorAmbient(u8 R, u8 G, u8 B);
    void SetColorDiffuse(u8 R, u8 G, u8 B);
    void SetColorSpecular0(u8 R, u8 G, u8 B);
    void SetColorSpecular1(u8 R, u8 G, u8 B);

    void SetPosition(f32 X, f32 Y, f32 Z);
    void SetDirection(f32 X, f32 Y, f32 Z);

    void EnableSpotLight(gdBool value);
    void SetSpotDirection(f32 X, f32 Y, f32 Z);
```

```
    void EnableDistanceAttenuation(gdBool value);
    void SetDistanceAttenuationScaleBias(
        f32 attenuationScale, f32 attenuationBias);
    nnResult UploadLookUpTableFloat(
        LightLookUpTableUploadId lightLutID, u32 lutStartIndex,
        const f32* valueData, const f32* deltaData, u32 dataCount);
    nnResult UploadLookUpTableNative(
        LightLookUpTableUploadId lightLutID, u32 lutStartIndex,
        const u32* data, u32 dataCount);
};
```

The following table shows which parameters are set by each function. For more information about how to specify arguments and what effect these parameters have, see the *CTR-SDK API Reference* and other documentation.

**Table 4-16. Parameters Set by Light Object Functions**

| Function | Parameters |
|---|---|
| `EnableLight` | Whether to enable light. Default value: `GD_FALSE` |
| `SetLightType` | Type of light source. Default value: `SOURCE_TYPE_POINT` |
| `EnableTwoSideDiffuse` | Whether to enable two-sided lighting. Default value: `GD_FALSE` |
| `EnableShadowed` | Whether to enable shadow attenuation. Default value: `GD_FALSE` |
| `EnableGeomFactor0` | Whether to enable geometry factor 0. Default value: `GD_FALSE` |
| `EnableGeomFactor1` | Whether to enable geometry factor 1. Default value: `GD_FALSE` |
| `SetColorAmbient` | Ambient light color. Default value: (0, 0, 0) |
| `SetColorDiffuse` | Diffuse light color. Default value: (255, 255, 255) |
| `SetColorSpecular0` | Color of specular light 0. Default value: (255, 255, 255) |
| `SetColorSpecular1` | Color of specular light 1. Default value: (255, 255, 255) |
| `SetPosition` | Point light source position. Default value: (0.0, 0.0, 0.0) |
| `SetDirection` | Directional light orientation. No default value. |
| `EnableSpotLight` | Whether to enable spotlights. Default value: `GD_FALSE` |
| `SetSpotDirection` | Spotlight direction. Default value: (0.0, 0.0, 0.0) |
| `EnableDistanceAttenuation` | Whether to enable distance attenuation. Default value: `GD_FALSE` |
| `SetDistanceAttenuationScaleBias` | Scale and bias for distance attenuation. Default values: `0.0`, `0.0`<br>A utility function (`Utils::ConvertStartEndToScaleBias`) has been provided to find the appropriate setting values from both a starting and ending distance. |

Lookup tables are loaded in the same way as they are for the lighting stage, although they are specified differently. **The lookup tables used by distance attenuation (DA) must be created to accept input values between 0.0 and 1.0.**

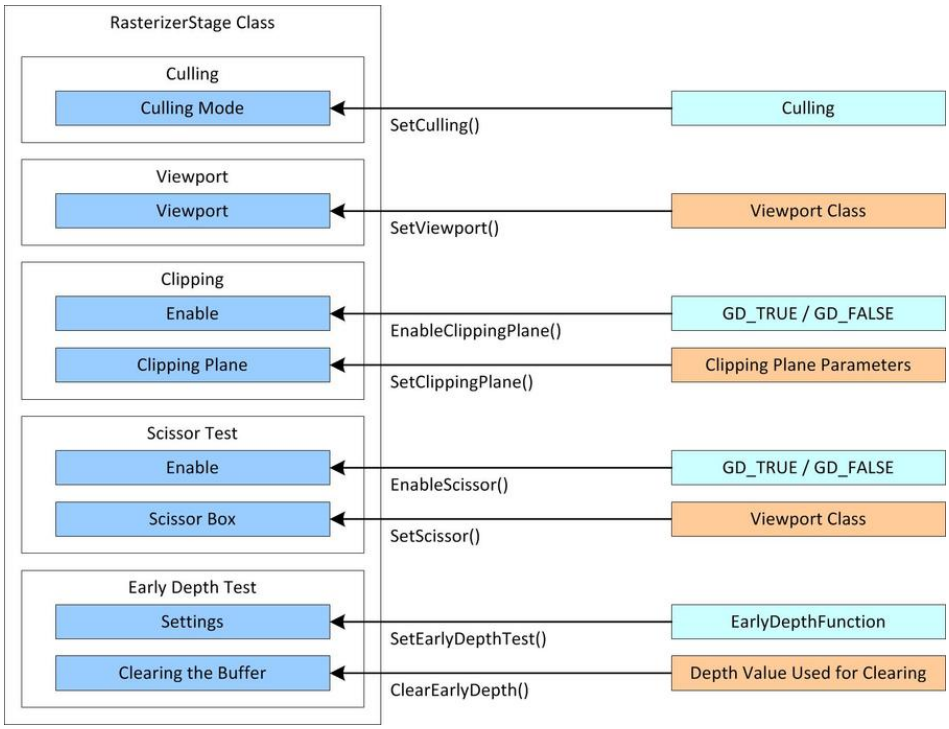
**Note:** Every function that configures lights is an immediate function, except for `EnableLight`, `EnableShadowed`, `EnableSpotLight`, and `EnableDistanceAttenuation`.

# 4.6. Rasterizer Stage

The rasterizer stage configures the process of rasterizing pixels that are output to the fragment pipeline. The `nn::gd::RasterizerStage` class defines functions that configure these settings.

The following diagram provides an overview of the settings made in the rasterizer stage.

**Figure 4-7. Overview of the Rasterizer Stage**




## 4.6.1. Culling Settings

Use the `nn::gd::RasterizerStage::SetCulling()` function to configure culling.

**Code 4-38. Function for Configuring Culling**

```
static void nn::gd::RasterizerStage::SetCulling(
        nn::gd::RasterizerStage::Culling culling);
```

Specify the culling method in the *`culling`* parameter. Clockwise culling (`CULLING_CLOCKWISE`) is specified by default.

## 4.6.2. Configuring the Viewport

Use the `nn::gd::RasterizerStage::SetViewport()` function to configure the viewport.

**Code 4-39. Functions for Configuring the Viewport**

```
class nn::gd::Viewport
{
    u32 m_X;
    u32 m_Y;
    u32 m_Width;
    u32 m_Height;

    Viewport();
    Viewport(u32 x, u32 y, u32 width, u32 height);
    void Set(u32 x, u32 y, u32 width, u32 height);
};

static void nn::gd::RasterizerStage::SetViewport(
        nn::gd::Viewport& viewport);
```

The `nn::gd::Viewport` class defines the viewport region. Note that the viewport has a different orientation than the LCDs. `Viewport(0, 0, 240, 320)` is set by default.

## 4.6.3. Clipping Settings

Use the following functions to configure clipping.

**Code 4-40. Functions for Configuring Clipping**

```
static void nn::gd::RasterizerStage::EnableClippingPlane(gdBool value);
static void nn::gd::RasterizerStage::SetClippingPlane(
        f32 param1, f32 param2, f32 param3, f32 param4);
```

By default, clipping is disabled (with `GD_FALSE`) and all parameters are set to `0`.

## 4.6.4. Scissor Test Settings

Use the following functions to configure the scissor test.

**Code 4-41. Functions for Configuring the Scissor Test**

```
static void nn::gd::RasterizerStage::EnableScissor(gdBool value);
static void nn::gd::RasterizerStage::SetScissor(
        nn::gd::Viewport& area);
```

By default, the scissor test is disabled (`GD_FALSE`) and `Viewport(0, 0, 0, 0)` is set as the scissor box.

## 4.6.5. Early Depth Test Settings

Use the following functions to configure the early depth test.

**Code 4-42. Functions for Configuring the Early Depth Test**

```
static void nn::gd::RasterizerStage::SetEarlyDepthTest(
        gdBool enable, nn::gd::RasterizerStage::EarlyDepthFunction func);
static void nn::gd::RasterizerStage::ClearEarlyDepth(f32 depthClearValue);
```

By default, the early depth test is disabled (`GD_FALSE` and `EARLYDEPTH_FUNCTION_LESS`) and `1.0` is set as the depth value to use for clearing.

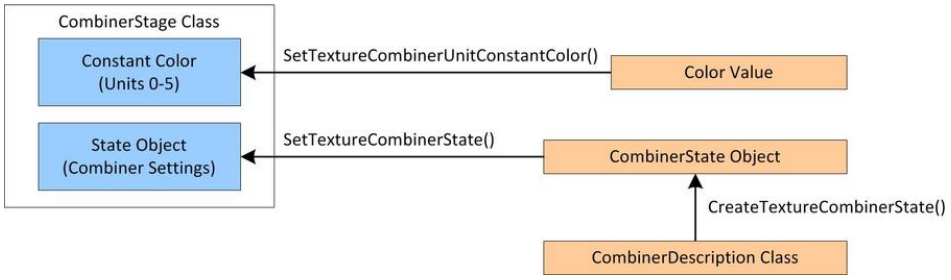
Block-32 mode must be set as the block mode when you use the early depth test. For more information, see Block Mode Settings in the *3DS Programming Manual: Advanced Graphics*.

# 4.7. Combiner Stage

The combiner stage configures (texture) combiners and combiner buffers. The `nn::gd::CombinerStage` class defines functions that configure these settings.

The following diagram provides an overview of the settings made in the combiner stage.

**Figure 4-8. Overview of the Combiner Stage**




## 4.7.1. Combiner Settings

The combiner stage creates a state (`CombinerState`) object that consolidates all combiner and combiner buffer settings. By switching this `CombinerState` object, you can change all combiner-related settings in one operation, except for the constant color.

`CombinerState` objects are created from a descriptor class (`CombinerDescription`) that defines operands and other settings.

**Code 4-43. Creating, Releasing, and Applying Combiner Settings to the Pipeline**

```
static nnResult nn::gd::CombinerStage::CreateTextureCombinerState(
        const nn::gd::CombinerDescription* description,
        nn::gd::CombinerState** textureCombinerState);
static nnResult nn::gd::CombinerStage::ReleaseTextureCombinerState(
        nn::gd::CombinerState* textureCombinerState);
static nnResult nn::gd::CombinerStage::SetTextureCombinerState(
        nn::gd::CombinerState* textureCombinerState);
```

Combiner settings are configured in the descriptor object (`CombinerDescription`) specified by the *`description`* parameter in advance. For more information about the descriptor class settings, see 4.9.5.1. Descriptor Class for BlendState Objects.

You can use the `nn::gd::CombinerStage::SetTextureCombinerState()` function to set a created `CombinerState` object in the combiner stage.

Use the `nn::gd::CombinerStage::ReleaseTextureCombinerStage()` function to release a

state object that is no longer needed. This also deallocates memory regions that were allocated internally when the `CombinerState` object was created.

Use the following function to set the constant color. You must set a constant color if it is used by combiner settings, because the constant color is undefined by default.

**Code 4-44. Setting the Constant Color**

```
static nnResult nn::gd::CombinerStage::SetTextureCombinerUnitConstantColor(
        nn::gd::CombinerStage::UnitId unit,
        u8 colorR, u8 colorG, u8 colorB, u8 colorA);
```

Because this is an immediate function, 3D commands are accumulated in the command buffer when it is run.

## 4.7.1.1. Descriptor Class for CombinerState Objects

This section introduces the members of the descriptor (`CombinerDescription`) class for `CombinerState` objects and provides setting-related cautions and information.

The `CombinerDescription` class contains settings for six combiner units. Because you will not necessarily configure every unit, this class is designed with member functions that set member variables for a specified unit.

**Code 4-45. Definition of the CombinerDescription Class**

```
class nn::gd::CombinerDescription
{
    struct BufferInput
    {
        nn::gd::CombinerStage::BufferInput m_Rgb1;
        nn::gd::CombinerStage::BufferInput m_Rgb2;
        nn::gd::CombinerStage::BufferInput m_Rgb3;
        nn::gd::CombinerStage::BufferInput m_Rgb4;
        nn::gd::CombinerStage::BufferInput m_Alpha1;
        nn::gd::CombinerStage::BufferInput m_Alpha2;
        nn::gd::CombinerStage::BufferInput m_Alpha3;
        nn::gd::CombinerStage::BufferInput m_Alpha4;
    };
    struct BufferInput m_BufferInput;

    void SetSourceRGB(
            nn::gd::CombinerStage::UnitId unit,
            nn::gd::CombinerStage::Source sourceRGB1,
            nn::gd::CombinerStage::Source sourceRGB2,
            nn::gd::CombinerStage::Source sourceRGB3);
    void SetOperandRGB(
            nn::gd::CombinerStage::UnitId unit,
            nn::gd::CombinerStage::OperandRgb opRGB1,
            nn::gd::CombinerStage::OperandRgb opRGB2,
            nn::gd::CombinerStage::OperandRgb opRGB3);
    void SetSourceAlpha(
            nn::gd::CombinerStage::UnitId unit,
            nn::gd::CombinerStage::Source sourceA1,
            nn::gd::CombinerStage::Source sourceA2,
            nn::gd::CombinerStage::Source sourceA3);
    void SetOperandAlpha(
            nn::gd::CombinerStage::UnitId unit,
            nn::gd::CombinerStage::OperandAlpha opA1,
            nn::gd::CombinerStage::OperandAlpha opA2,
            nn::gd::CombinerStage::OperandAlpha opA3);
    void SetCombineRGB(
            nn::gd::CombinerStage::UnitId unit,
            nn::gd::CombinerStage::CombineRgb combineRgb);
    void SetCombineAlpha(
            nn::gd::CombinerStage::UnitId unit,
```

```
                nn::gd::CombinerStage::CombineAlpha combineAlpha);
    void SetScaleRGB(
                nn::gd::CombinerStage::UnitId unit,
                nn::gd::CombinerStage::Scale scaleRgb);
    void SetScaleAlpha(
                nn::gd::CombinerStage::UnitId unit,
                nn::gd::CombinerStage::Scale scaleAlpha);
    void SetBufferColor(u8 colorR, u8 colorG, u8 colorB, u8 colorA);
    void SetCombinerInUse(nn::gd::CombinerStage::UnitId unit, gdBool mode);

    CombinerDescription();
    void ToDefault();
};
```

### Input Source Settings (SetSourceRGB, SetSourceAlpha)

These functions set input sources 0 to 2 for the specified combiner unit.

### Operand Settings (SetOperandRGB, SetOperandAlpha)

These functions set operands 0 to 2 for the specified combiner unit.

### Combiner Function Settings (SetCombineRGB, SetCombineAlpha)

These functions set the combiner function for the specified combiner unit.

### Scaling Value Settings (SetScaleRGB, SetScaleAlpha)

These functions set the scaling values for the specified combiner unit.

### Constant Color for Combiner Buffer 0 Setting (SetBufferColor)

This function sets the constant color for combiner buffer 0.

### Combiner Buffer Input Source (m_BufferInput)

Sets the input source for combiner buffers 1 to 4. No setter functions are defined for it.

### Configuring How Combiner Units Are Used (SetCombinerInUse)

This function either enables (`GD_TRUE`) or disables (`GD_FALSE`) the settings for the specified combiner unit. If you do not enable settings for a combiner unit, it uses its default settings.

### Default Settings (ToDefault)

You can call the `ToDefault()` function to change all members to their default values. This is also done in the constructor.

**Table 4-17. Default Combiner Settings**

| Member | Default Settings |
|---|---|
| `SetSourceRGB`<br>`SetSourceAlpha` | `SOURCE_CONSTANT` (all input sources for unit 0)<br>`SOURCE_PREVIOUS` (all input sources for units 1 to 5) |

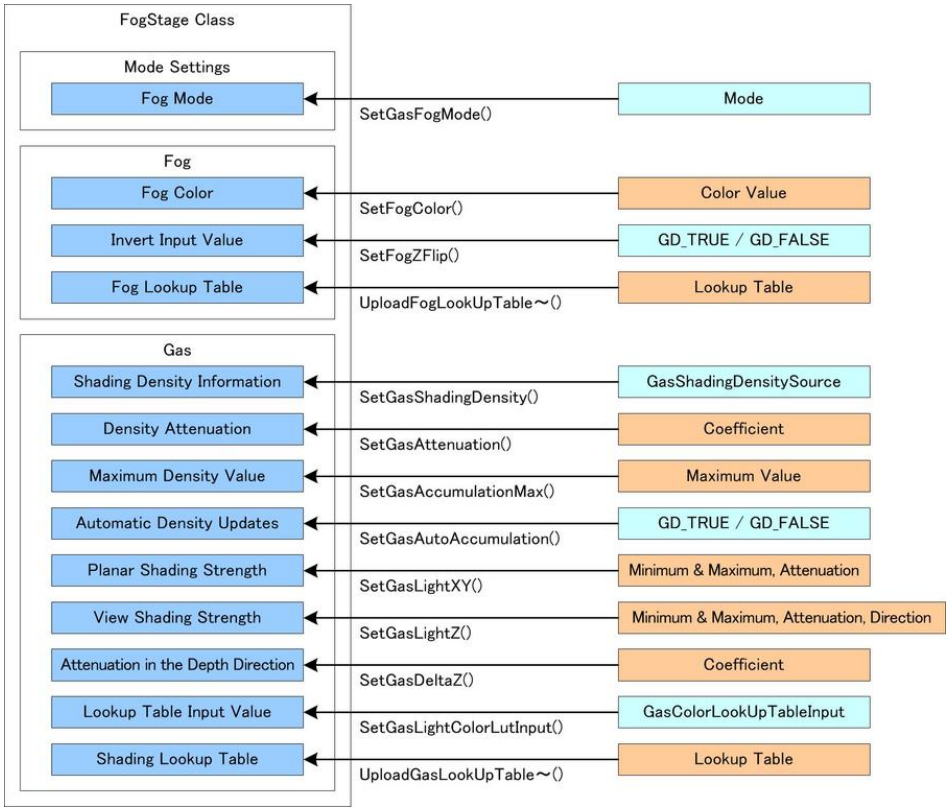
| | |
|---|---|
| SetOperandRGB | OPERAND_RGB_SRC_COLOR (all operands) |
| SetOperandAlpha | OPERAND_ALPHA_SRC_ALPHA (all operands) |
| SetCombineRGB | COMBINE_RGB_REPLACE (all units) |
| SetCombineAlpha | COMBINE_ALPHA_REPLACE (all units) |
| SetScaleRGB<br>SetScaleAlpha | SCALE_1 (all units) |
| SetBufferColor | (0,0,0,0) |
| m_BufferInput | INPUT_PREVIOUS_BUFFER (all inputs) |
| SetCombinerInUse | GD_FALSE (all units) |

# 4.8. Fog Stage

The fog stage handles all settings related to rendering fog and gas. The nn::gd::FogStage class defines functions that configure these settings.

The following diagram provides an overview of the settings made in the fog stage.

**Figure 4-9. Overview of the Fog Stage**




## 4.8.1. Fog Mode

Use the nn::gd::FogStage::SetGasFogMode() function to switch the mode for GPU fog

features.

**Code 4-46. Switching the Fog Mode**

```
static void nn::gd::FogStage::SetGasFogMode(nn::gd::FogStage::Mode mode);
```

You can specify one of the following values in the *mode* parameter.

**Table 4-18. Specifying the Fog Mode**

| Definition | Description |
|---|---|
| FogStage::MODE_NONE | Do not use fog features. (Default.) |
| FogStage::MODE_FOG | Use fog features normally in fog mode. |
| FogStage::MODE_GAS | Use fog features in gas mode. |

## 4.8.2. Fog Settings

Functions have been provided for setting each of the fog parameters.

**Code 4-47. Functions for Setting Fog Parameters**

```
class nn::gd::FogStage
{
    static void SetFogColor(u8 R, u8 G, u8 B);
    static void SetFogZFlip(gdBool zFlip);
};
```

The following table shows which parameters are set by each function. For more information about how to specify arguments and what effect these parameters have, see the *CTR-SDK API Reference* and other documentation.

**Table 4-19. Parameters Configured by Functions in the Fog Stage**

| Function | Parameters |
|---|---|
| SetFogColor | Fog color. Default value: (0, 0, 0) |
| SetFogZFlip | Whether to invert input depth value. Default value: GD_FALSE |

**Note:** For more information about how to load lookup tables, see 4.8.4. Loading Lookup Tables Used by the Fog Stage.

## 4.8.3. Gas Settings

Functions have been provided for setting each of the gas parameters.

**Code 4-48. Functions for Setting Gas Parameters**

```
class nn::gd::FogStage
```

```
{
    static void SetGasShadingDensity(
        GasShadingDensitySource shadingDensitySrc);
    static void SetGasAttenuation(f32 attenuation);
    static void SetGasAccumulationMax(f32 maxAccumulation);
    static void SetGasAutoAccumulation(gdBool enableAutoAccumulation);
    static void SetGasLightXY(
        u8 lightMinimum, u8 lightMaximum, u8 lightAttenuation);
    static void SetGasLightZ(
        u8 scattMinimum, u8 scattMaximum, u8 scattAttenuation, u8 lz);
    static void SetGasDeltaZ(f32 deltaZ);
    static void SetGasLightColorLutInput(
        GasColorLookUpTableInput colorLutInput);
};
```

The following table shows which parameters are set by each function. For more information about how to specify arguments and what effect these parameters have, see the *CTR-SDK API Reference* and other documentation.

**Table 4-20. Gas Parameters Configured by Functions in the Fog Stage**

| Function | Parameters |
|---|---|
| `SetGasShadingDensity` | Density information used for shading. Default value: `GAS_SHADING_DENSITY_SOURCE_PLAIN` |
| `SetGasAttenuation` | Density attenuation coefficient to use when calculating alpha value for shading. Default value: `1.0` |
| `SetGasAccumulationMax` | The reciprocal of the maximum density information value. Default value: `1.0` |
| `SetGasAutoAccumulation` | Whether to automatically calculate the reciprocal of the maximum density information value. Default value: `GD_FALSE` |
| `SetGasLightXY` | Factors used to control planar shading: the minimum intensity, maximum intensity, and density attenuation. Default value: `(0,0,0)` |
| `SetGasLightZ` | Factors used to control view shading: the minimum intensity, maximum intensity, density attenuation, and effect in the view direction. Default value: `(0,0,0,0)` |
| `SetGasDeltaZ` | Attenuation coefficient in the depth direction. Default value: `10.0` |
| `SetGasLightColorLutInput` | Input value to the shading lookup table. Default value: `GAS_COLOR_INPUT_LIGHT_FACTOR` |

**Note:** For more information about how to load lookup tables, see 4.8.4. Loading Lookup Tables Used by the Fog Stage.

## 4.8.4. Loading Lookup Tables Used by the Fog Stage

There are two ways to load the lookup tables used for coefficients and shading in the fog stage: The first is through functions (with the `Float` suffix) that accept arrays of floating-point numbers. The second is through functions (with the `Native` suffix) that accept arrays of data to be written unchanged to the GPU registers. Because these are immediate functions, data is written to the 3D command buffer when they are run.

**Code 4-49. Functions for Loading Lookup Tables Used by the Fog Stage**

```
class nn::gd::FogStage
{
    static nnResult UploadFogLookUpTableNative(
```

```
        u32 lutStartIndex, u32* data, u32 countData);
    static nnResult UploadFogLookUpTableFloat(
        u32 lutStartIndex, const f32* dataValue, const f32* dataDelta,
        u32 countData);
    static nnResult UploadGasLookUpTableNative(
        u32 lutStartIndex, u32* data, u32 countData);
    static nnResult UploadGasLookUpTableFloat(
        u32 lutStartIndex, f32* dataValue, f32* dataDelta, u32 countData);
};
```

You can use *lutStartindex* and *countData* values to partially overwrite a lookup table. However, an error occurs if the total exceeds the maximum number of elements in the following table.

To load a lookup table using arrays of floating-point numbers, you must provide one array for the data (the *dataValue* parameter) and one for the deltas (the *dataDelta* parameter). However, note that because each element in the shading lookup table is generated from three elements in these source arrays (which store the color components in RGB order), *dataValue* and *dataDelta* must have three times the number of elements specified in *countData*.

If you load any lookup table using an array that has been converted into native format, you only need to provide one array because both the data and the delta have been combined into a single element. However, the native and floating-point arrays used to load the shading lookup table have different maximum sizes.

The following table shows the format of and maximum number of elements in the arrays used by each function to load a lookup table.

**Table 4-21. Functions for Loading Lookup Tables Used by the Fog Stage**

| Function | Lookup Table | Format | Maximum Number of Elements |
|---|---|---|---|
| UploadFogLookUpTableFloat | Fog Coefficients | Floating-point numbers | 128 |
| UploadFogLookUpTableNative | Fog Coefficients | Native | 128 |
| UploadGasLookUpTableFloat | Shading | Floating-point numbers | 8 |
| UploadGasLookUpTableNative | Shading | Native | 16 |

## 4.8.4.1. Helper Functions for Loading Lookup Tables

Helper functions have been provided for converting arrays of floating-point numbers into native format. Functions that load lookup tables from arrays of floating-point numbers internally convert the arrays into native format. This is expensive and not suited for runtime use. We recommend that you use some other strategy, such as providing data that has already been converted into native format or using the helper functions to convert the data when the lookup table is loaded for the first time.

**Code 4-50. Helper Functions for Loading Lookup Tables**

```
class nn::gd::FogStage::Helper
{
    static nnResult ConvertFogLookUpTableDataFloatToNative(
        const f32* dataValue, const f32* dataDelta, u32 countData,
        u32 *__restrict dest);
    static nnResult ConvertGasLookUpTableDataFloatToNative(
        const f32* dataValue, const f32* dataDelta, u32 countData,
```

```
            u32 *__restrict dest);
};
```

Both helper functions provided by the fog stage generate a single array (the *dest* parameter) from a data array and a delta array.

Each data element is converted into an 11-bit unsigned fixed-point number with 11 fractional bits. Each delta is converted into a 13-bit signed (two's complement) fixed-point number with 11 fractional bits. These are stored in bits [23:13] and [12:0], respectively, of the data generated for the fog coefficient lookup table.

Data is generated for the shading lookup table in a special format. Specifically, note that the three elements of the source array (in R/G/B order) are converted into a single data value. Conversely, each element of the array `dest` does not contain both data and delta values. Instead, the first eight elements of the array store delta values, and the second eight elements store data values.

The first eight elements are generated from the deltas. The deltas for the R, G, and B components are converted into 8-bit signed integers that are packed into bits [7:0], [15:8], and [23:16] respectively.

The last eight elements are generated from the data. The data for the R, G, and B components are converted into 8-bit unsigned fixed-point numbers (with no fractional bits) that are packed into bits [7:0], [15:8], and [23:16] respectively.

**Note:** For more information about how these values are converted, see the *3DS Programming Manual: Advanced Graphics*.

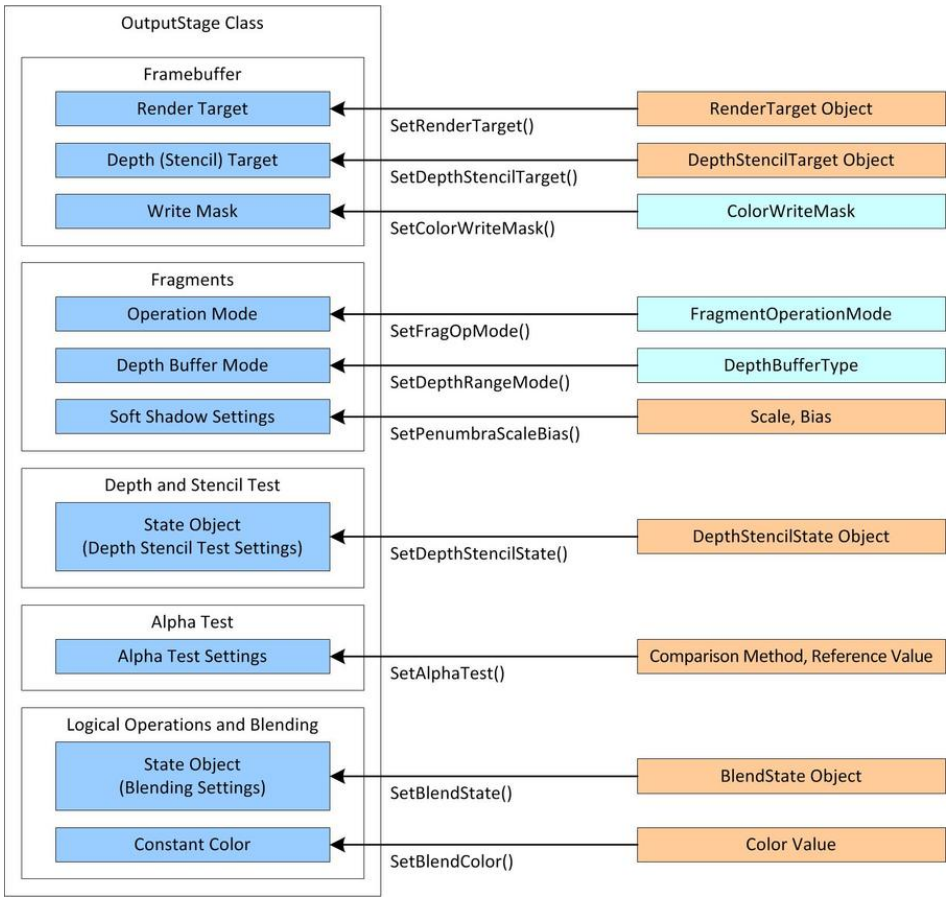
The GD library provides utility functions for these conversions.

## 4.9. Output Stage

The output stage configures blending, the depth test, the stencil test, the target buffer, and other settings related to pixel output. The `nn::gd::OutputStage` class defines functions that configure these settings.

The following diagram provides an overview of the settings made in the output stage.

**Figure 4-10. Overview of the Output Stage**




## 4.9.1. Fragment Operation Mode

You can use the `nn::gd::OutputStage::SetFragOpMode()` function to switch the fragment operation mode.

**Code 4-51. Function for Setting the Fragment Operation Mode**

```
static void nn::gd::OutputStage::SetFragOpMode(
        nn::gd::OutputStage::FragmentOperationMode fragOpMode);
```

You can specify one of the following values in the *`fragOpMode`* parameter.

**Table 4-22. Fragment Operation Modes**

| Definition | Description |
|---|---|
| FRAGMENT_OPERATION_MODE_FRAGMENT_COLOR | Normal fragment processing (this is the default value). |
| FRAGMENT_OPERATION_MODE_GAS_ACCUMULATION | Fragment processing for the shadow accumulation path. |
| FRAGMENT_OPERATION_MODE_SHADOW_MAP | Fragment processing for rendering gas density information. |

## 4.9.2. Framebuffers

The GD library uses RenderTarget and DepthStencilTarget objects to operate on the framebuffer (color and depth stencil buffer). Both objects are created from a single Texture2DResource object and a descriptor class (RenderTargetDescription or DepthStencilTargetDescription,). If the Texture2DResource object has mipmap data, you can use descriptor class settings to specify which levels of mipmap data to use when you create the object.

You can use the following functions to create, release, and apply a framebuffer object (a RenderTarget or DepthStencilTarget object) to the pipeline.

**Code 4-52. Creating, Releasing, and Applying a Framebuffer Object to the Pipeline**

```
static nnResult nn::gd::OutputStage::CreateRenderTarget(
        const nn::gd::Texture2DResource* texture2DResource,
        const nn::gd::RenderTargetDescription* desc,
        nn::gd::RenderTarget** renderTarget);
static nnResult nn::gd::OutputStage::ReleaseRenderTarget(
        nn::gd::RenderTarget* renderTarget);
static nnResult nn::gd::OutputStage::SetRenderTarget(
        const nn::gd::RenderTarget* renderTarget);

static nnResult nn::gd::OutputStage::CreateDepthStencilTarget(
        const nn::gd::Texture2DResource* texture2DResource,
        const nn::gd::DepthStencilTargetDescription* desc,
        nn::gd::DepthStencilTarget** depthStencil);
static nnResult nn::gd::OutputStage::ReleaseDepthStencilTarget(
        nn::gd::DepthStencilTarget* depthStencil);
static nnResult nn::gd::OutputStage::SetDepthStencilTarget(
        const nn::gd::DepthStencilTarget* depthStencil);
```

Functions that have the RenderTarget suffix affect the color buffer. Functions that have the DepthStencilTarget suffix affect the depth (stencil) buffer.

Configure framebuffer settings in the corresponding descriptor object (RenderTargetDescription or DepthStencilTargetDescription) specified in the *desc* parameter. For more information about the descriptor class settings, see 4.9.2.1. Descriptor Class for Framebuffer Objects.

Only a limited number of pixel formats can be used by a Texture2DResource object in the framebuffer. For more information, see 3.1.1. Creating a Texture2DResource Object.

After you have created a framebuffer object, you can set it as the framebuffer with the nn::gd::OutputStage::Set∗() functions.

Use the nn::gd::OutputStage::Release∗() functions to release framebuffer objects that are no longer needed. Only management data is released when a framebuffer object is released. Texture2DResource objects are not released.

## 4.9.2.1. Descriptor Class for Framebuffer Objects

This section introduces the members of the descriptor classes (RenderTargetDescription and DepthStencilTargetDescription) for framebuffer classes (RenderTarget and DepthStencilTarget) and provides setting-related cautions and information.

**Code 4-53. Descriptor Class Definitions for Framebuffer Objects**

```
class nn::gd::RenderTargetDescription
{
    u32 m_MipLevelIndex;
```

```
};
class nn::gd::DepthStencilTargetDescription
{
    u32 m_MipLevelIndex;
};
```

#### Mipmap Level Index (m_MipLevelIndex)

This is the mipmap level of the `Texture2DResource` object used to create a framebuffer object.

## 4.9.2.2. Getting Detailed Information About Framebuffer Objects

Detailed information (the `TargetProperties` class) includes framebuffer-related information that is based on the content of the `Texture2DResource` object specified when the framebuffer was created.

The `RenderTarget` and `DepthStencilTarget` objects use different functions to get detailed information, but they both do so using the `TargetProperties` class.

**Code 4-54. Getting Detailed Information About Framebuffer Objects**

```
static nnResult nn::gd::OutputStage::GetRenderTargetProperties(
        const nn::gd::RenderTarget* renderTarget,
        nn::gd::TargetProperties* properties);
static nnResult nn::gd::OutputStage::GetDepthStencilTargetProperties(
        const nn::gd::DepthStencilTarget* depthStencil,
        nn::gd::TargetProperties* properties);
```

Although the width, height, mipmap levels, and starting address of resource data may change with the mipmap levels specified when an object was created, they are generally the same as the detailed information for the `Texture2DResource` object.

**Code 4-55. Definition of the TargetProperties Class**

```
class nn::gd::TargetProperties
{
    u32 m_Width;
    u32 m_Height;
    u32 m_PixelSize;
    nn::gd::Resource::NativeFormat m_Format;
    nn::gd::Memory::MemoryLayout m_MemLayout;
    nn::gd::Memory::MemoryLocation m_MemLocation;
    u8* m_MemAddr;
};
```

## 4.9.2.3. Clearing the Framebuffer

You can clear the color buffer and depth (stencil) buffer using a specified value.

**Code 4-56. Clearing the Framebuffer**

```
static nnResult nn::gd::Memory::ClearTargets(
        const nn::gd::RenderTarget* renderTarget,
        const nn::gd::DepthStencilTarget* depthStencilTarget,
        const u8 ColorRGBA[4], float depth, u8 stencil);
```

The framebuffer is actually cleared when the command list is executed because this function internally calls the `nngxAddMemoryFillCommand()` function.

### 4.9.2.4. Color Mask

Using the `nn::gd::OutputStage::SetColorWriteMask()` function, you can change color mask settings and control what is written to the color buffer.

**Code 4-57. Function for Setting the Color Mask**

```
static void nn::gd::OutputStage::SetColorWriteMask(u32 mask);
```

Specify a bitwise OR of the following flags in the *mask* parameter.

**Table 4-23. Color Mask Flags**

| Definition | Description |
|---|---|
| COLOR_WRITE_MASK_ENABLE_RED | Enables write operations for the red component. |
| COLOR_WRITE_MASK_ENABLE_GREEN | Enables write operations for the green component. |
| COLOR_WRITE_MASK_ENABLE_BLUE | Enables write operations for the blue component. |
| COLOR_WRITE_MASK_ENABLE_ALPHA | Enables write operations for the alpha component. |
| COLOR_WRITE_MASK_ENABLE_ALL | Enables write operations for all components. (This value is the default.) |

## 4.9.3. Depth and Stencil Tests

By creating a state (`DepthStencilState`) object and then switching the `DepthStencilState` object that is configured in the output stage, you can change all settings related to the depth and stencil test in one operation.

You can create, release, and apply a `DepthStencilState` object to the pipeline using the following functions.

**Code 4-58. Creating, Releasing, and Applying a DepthStencilState Object to the Pipeline**

```
static nnResult nn::gd::OutputStage::CreateDepthStencilState(
        const nn::gd::DepthStencilStateDescription* desc,
        nn::gd::DepthStencilState** DepthStencilState);
static nnResult nn::gd::OutputStage::ReleaseDepthStencilState(
        nn::gd::DepthStencilState* DepthStencilState);
static nnResult nn::gd::OutputStage::SetDepthStencilState(
        const nn::gd::DepthStencilState* state, u8 stencilRef);
```

Configure the depth and stencil test in a descriptor class (`DepthStencilStateDescription`) passed in the *desc* parameter. For more information about the descriptor class settings, see 4.9.3.1. Descriptor Class for DepthStencilState Objects.

After you have created a state object, you can apply it to the pipeline using the `nn::gd::OutputStage::SetDepthStencilState()` function. Specify the reference value to

use in the stencil test in the *stencilRef* parameter.

Use the nn::gd::OutputStage::ReleaseDepthStencilState() function to release a state object that is no longer needed. This also releases memory regions that were allocated internally when the BlendState object was created as a state object.

## 4.9.3.1. Descriptor Class for DepthStencilState Objects

This section introduces the members of the descriptor (DepthStencilStateDescription) class for DepthStencilState objects and provides setting-related cautions and information.

**Code 4-59. Definition of the DepthStencilStateDescription Class**

```
class nn::gd::DepthStencilStateDescription
{
    gdBool                                  m_DepthEnable;
    gdBool                                  m_StencilEnable;
    u8                                      m_StencilWriteMask;
    u8                                      m_StencilReadMask;
    nn::gd::OutputStage::DepthWriteMask     m_DepthMask;
    nn::gd::OutputStage::DepthFunction      m_DepthFunc;
    nn::gd::OutputStage::StencilFunction    m_StencilFunc;
    nn::gd::OutputStage::StencilOperation   m_StencilFail;
    nn::gd::OutputStage::StencilOperation   m_StencilZFail;
    nn::gd::OutputStage::StencilOperation   m_StencilZPass;

    DepthStencilStateDescription();
    void ToDefault();
};
```

### Enabling the Depth Test (m_DepthEnable)

The depth test is enabled if this is GD_TRUE.

### Write Mask for the Depth Test (m_DepthMask)

Sets the write mask for the depth test.

### Comparison Function for the Depth Test (m_DepthFunc)

Sets the comparison function for the depth test.

### Enabling the Stencil Test (m_StencilEnable)

The stencil test is enabled if this is GD_TRUE.

### Write Mask for the Stencil Test (m_StencilWriteMask)

Sets the write mask for the stencil test.

### Read Mask for the Stencil Test (m_StencilReadMask)

Sets the read mask for the stencil test.

#### Comparison Function for the Stencil Test (m_StencilFunc)

Sets the comparison function for the stencil test.

#### Operation When the Stencil Test Fails (m_StencilFail)

Sets the operation to run when the stencil test fails.

#### Operation When the Stencil Test Passes (m_StencilZFail)

Sets the operation to run when the stencil test passes, but the depth test fails.

#### Operation When Both the Depth and Stencil Test Pass (m_StencilZPass)

Sets the operation to run when both the depth and stencil tests pass.

#### Default Settings (ToDefault)

You can call the `ToDefault()` function to change all members to their default values. This is also done in the constructor.

**Table 4-24. Default Settings for the DepthStencilStateDescription Class**

| Member | Default Settings |
|---|---|
| `m_DepthEnable` | `GD_FALSE` |
| `m_DepthMask` | `DEPTH_WRITE_MASK_ALL` |
| `m_DepthFunc` | `DEPTH_FUNCTION_LESS` |
| `m_StencilEnable` | `GD_FALSE` |
| `m_StencilWriteMask`<br>`m_StencilReadMask` | `0xFF` |
| `m_StencilFunc` | `STENCIL_FUNCTION_NEVER` |
| `m_StencilFail`<br>`m_StencilZFail`<br>`m_StencilZPass` | `STENCIL_OPERATION_KEEP` |

## 4.9.4. Alpha Test

Use the `nn::gd::OutputStage::SetAlphaTest()` function to configure the alpha test.

**Code 4-60. Configuring the Alpha Test**

```
static void nn::gd::OutputStage::SetAlphaTest(
        gdBool enable, nn::gd::OutputStage::AlphaFunction func, u8 alphaRef);
```

To enable the alpha test, specify `GD_TRUE` in the *`enable`* parameter, a comparison function in the *`func`* parameter, and a reference value in the *`alphaRef`* parameter. The default settings for these parameters are `GD_FALSE`, `ALPHA_FUNCTION_NEVER`, and `0`, respectively.

## 4.9.5. Logical Operations and Blending

By creating a state (`BlendState`) object and then switching the `BlendState` object that is configured in the output stage, you can change all settings related to logical operations and blending in one operation.

You can create, release, and apply a `BlendState` object to the pipeline using the following functions.

**Code 4-61. Creating, Releasing, and Applying a BlendState Object to the Pipeline**

```
static nnResult nn::gd::OutputStage::CreateBlendState(
        const nn::gd::BlendStateDescription* desc,
        nn::gd::BlendState** blendState);
static nnResult nn::gd::OutputStage::ReleaseBlendState(
        nn::gd::BlendState* blendState);
static nnResult nn::gd::OutputStage::SetBlendState(
        const nn::gd::BlendState* blendState);
```

Configure logical operations and blending in a descriptor class (`BlendStateDescription`) that is specified in the *`desc`* parameter. For more information about the descriptor class settings, see 4.9.5.1. Descriptor Class for BlendState Objects.

You can use the `nn::gd::OutputStage::SetBlendState()` function to apply a created `BlendState` object to the pipeline.

Use the `nn::gd::OutputStage::ReleaseBlendState()` function to release a state object that is no longer needed. This also releases memory regions that were allocated internally when the `BlendState` object was created as a state object.

Use the following function to set the blend color. The default blend color is `(0, 0, 0, 0)`.

**Code 4-62. Setting the Blend Color**

```
static void nn::gd::OutputStage::SetBlendColor(u8 R, u8 G, u8 B, u8 A);
```

### 4.9.5.1. Descriptor Class for BlendState Objects

This section introduces the members of the descriptor (`BlendStateDescription`) class for `BlendState` objects and provides setting-related cautions and information.

**Code 4-63. Definition of the BlendStateDescription Class**

```
class nn::gd::BlendStateDescription
{
    nn::gd::OutputStage::BlendType m_BlendType;
    nn::gd::OutputStage::LogicOperator m_LogicOp;
    nn::gd::OutputStage::BlendFunction m_SrcRgb;
    nn::gd::OutputStage::BlendFunction m_DstRgb;
    nn::gd::OutputStage::BlendFunction m_SrcAlpha;
    nn::gd::OutputStage::BlendFunction m_DstAlpha;
    nn::gd::OutputStage::BlendEquation m_EqRgb;
    nn::gd::OutputStage::BlendEquation m_EqAlpha;

    BlendStateDescription();
    void ToDefault();

    void SetBlendFunc(
        nn::gd::OutputStage::BlendFunction src,
```

```
            nn::gd::OutputStage::BlendFunction dst,
            nn::gd::OutputStage::BlendEquation eq);
        void SetBlendFunc(
            nn::gd::OutputStage::BlendFunction srcRgb,
            nn::gd::OutputStage::BlendFunction dstRgb,
            nn::gd::OutputStage::BlendFunction srcAlpha,
            nn::gd::OutputStage::BlendFunction dstAlpha,
            nn::gd::OutputStage::BlendEquation eqRgb,
            nn::gd::OutputStage::BlendEquation eqAlpha);

        void SetBlendMode_DefaultBlending();
        void SetBlendMode_NoBlend();

        void SetLogicOperatorMode(nn::gd::OutputStage::LogicOperator logicOp);
        void SetLogicOperatorMode_Default();
    };
```

### Choice Between Logical Operations and Blending (m_BlendType)

Set to `BLEND_TYPE_LOGICOP` when logical operations are used or `BLEND_TYPE_BLENDING` when blending is used.

### Logical Operation (m_LogicOp)

Indicates which logical operation to use. You can set this value using the `SetLogicOperatorMode()` function.

Set to `LOGIC_OPERATOR_COPY` by the `SetLogicOperatorMode_Default()` function.

### Blend Weight Coefficients (m_SrcRgb, m_DstRgb, m_SrcAlpha, m_DstAlpha)

These are the blend weight coefficients. There are settings for the RGB and alpha components in both the source and destination.

### Blend Equations (m_EqRgb, m_EqAlpha)

These are the blend equations. There is a setting for both the RGB and alpha components.

### Batch Blend Settings (SetBlendFunc, SetBlendMode_NoBlend)

The `SetBlendFunc()` function configures all blending operations at the same time. You can either configure the RGB and alpha components together or separately.

The `SetBlendMode_NoBlend()` function changes settings so that there is no blending (the source is output unchanged).

**Table 4-25. Blend Settings Changed by SetBlendMode_NoBlend**

| Member | Setting |
|---|---|
| `m_SrcRgb, m_SrcAlpha` | `BLEND_FUNCTION_ONE` |
| `m_DstRgb, m_DstAlpha` | `BLEND_FUNCTION_ZERO` |
| `m_EqRgb, m_EqAlpha` | `BLEND_EQUATION_ADD` |

### Default Settings (ToDefault)

You can call the `ToDefault()` function to change all members to their default values. This is also done in the constructor.

Blending is used by default, and the settings are the same as those configured by the `SetBlendMode_NoBlend()` function.

### 4.9.6. W-Buffer and Polygon Offset

Use the `nn::gd::OutputStage::SetDepthRangeMode()` function to configure the w-buffer and polygon offset.

**Code 4-64. Setting the W-Buffer and Polygon Offset**

```
static void nn::gd::OutputStage::SetDepthRangeMode(
        nn::gd::OutputStage::DepthBufferType type,
        f32 depthRangeNear, f32 depthRangeFar, s32 offset);
```

Specify `DEPTHBUFFER_LINEAR` in the *`type`* parameter to use the w-buffer feature or specify `DEPTHBUFFER_RECIPROCAL_FUNCTION` to use the normal range of depth values instead.

Specify the near and far depth values in the *`depthRangeNear`* and *`depthRangeFar`* parameters when you do not use the w-buffer. Specify `0` in *`depthRangeNear`* and a scale value in *`depthRangeFar`* when you do use the w-buffer.

When you use a polygon offset, specify the offset value in the *`offset`* parameter.

### 4.9.7. Soft Shadows

Use the `nn::gd::OutputStage::SetPenumbraScaleBias()` function to set the scale and bias for soft shadows. Because this is an immediate function, 3D commands are accumulated in the command buffer when it is run.

**Code 4-65. Setting Soft Shadows**

```
static void nn::gd::OutputStage::SetPenumbraScaleBias(f32 Scale, f32 Bias);
```



# 5. Layout Conversion

The `nn::gd::Memory` class defines functions that convert layouts from block format to linear format and vice versa.

# 5.1. Converting From Block Format to Linear Format

The following functions are provided for converting from block format to linear format.

**Code 5-1. Functions for Converting From Block Format to Linear Format**

```
static nnResult nn::gd::Memory::CopyTexture2DResourceBlockToLinear(
        const nn::gd::Texture2DResource* source,
        s32 srcMipLevelIndex, u32 srcOffsetY, s32 srcCountRow,
        const nn::gd::Texture2DResource* dest,
        s32 dstMipLevelIndex, u32 dstOffsetY,
        nn::gd::Memory::DownScalingMode downScalingMode, gdBool yFlip);
static nnResult nn::gd::Memory::CopyTexture2DResourceBlockToLinear(
        const nn::gd::Texture2DResource* source,
        s32 srcMipLevelIndex, u32 srcOffsetY,
        u8* dstAddr, u32 dstWidth, u32 dstHeight, u32 dstFormat,
        nn::gd::Memory::DownScalingMode downScalingMode, gdBool yFlip);
```

The resource data in the `Texture2DResource` object specified for the *`source`* parameter is converted from block format to linear format and then transferred. You can use the *`srcMipLevelIndex`* parameter to specify which mipmap level to convert. If you specify a value of –1, the largest mipmap level of the `Texture2DResource` object is used.

Use the *`srcOffsetY`* parameter to specify the y-coordinate at which to start converting. Specify the raw resource data coordinates. These functions take care of addressing differences.

When you call the function that takes a `Texture2DResource` object as the destination, use the *`srcCountRow`* parameter to specify the number of lines to convert. If you specify a value of –1, the height of the `Texture2DResource` object is used. Use the *`dstMipLevelIndex`* parameter and the *`dstOffsetY`* parameter to specify the mipmap level and starting y-coordinate, respectively, at the destination.

When you call the function that takes an address as the destination, specify the starting address in the *`dstAddr`* parameter, the width in the *`dstWidth`* parameter, the height in the *`dstHeight`* parameter, and the pixel format in the *`dstFormat`* parameter.

Configure anti-aliasing using the *`downScalingMode`* parameter. To enable vertical flipping, specify `GD_TRUE` in the *`yFlip`* parameter.

Data is actually converted when the command list is executed, because these functions call the `nngxAddB2LTransferCommand()` function.

These functions, which convert from block format to linear format, are mainly used when the color buffer is transferred to the display buffer. For sample code, see 2.7. Displaying the Rendered Results.

# 5.2. Converting From Linear Format to Block Format

The following functions have been provided for converting from linear format to block format.

**Code 5-2. Functions for Converting From Linear Format to Block Format**

```
static nnResult nn::gd::Memory::CopyTexture2DResourceLinearToBlock(
        const nn::gd::Texture2DResource* source,
        s32 srcMipLevelIndex, u32 srcOffsetY, s32 srcCountRow,
        const nn::gd::Texture2DResource* dest,
```

```
        s32 dstMipLevelIndex, u32 dstOffsetY);
static nnResult nn::gd::Memory::CopyTexture2DResourceLinearToBlock(
        u8* srcAddr, u32 width, u32 height,
        const nn::gd::Texture2DResource* dest,
        s32 dstMipLevelIndex, u32 dstOffsetY);
```

The original data is converted from linear format to block format and then transferred to the `Texture2DResource` object specified for the *`dest`* parameter. You can use the *`dstMipLevelIndex`* parameter to specify the mipmap level in which to receive the converted data. If you specify a value of –1, the largest mipmap level of the `Texture2DResource` object is used. Use *`dstOffsetY`* to specify the y-coordinate at which to start receiving data. Specify the raw resource data coordinates. These functions take care of addressing differences.

When you call the function that takes a `Texture2DResource` object as the source, use the *`srcCountRow`* parameter to specify the number of lines to convert. If you specify a value of –1, the height of the `Texture2DResource` object is used. Use the *`srcMipLevelIndex`* parameter and the *`srcOffsetY`* parameter to specify the mipmap level and starting y-coordinate, respectively, at the source.

When you call the function that takes an address as the source, specify the starting address in the *`srcAddr`* parameter, the width in the *`width`* parameter, and the height in the *`height`* parameter.

Data is actually converted when the command list is executed, because these functions call the `nngxAddL2BTransferCommand()` function.



# 6. Packet Recording

Using the packet recording feature, you can record a group of command packets and then replay them later. This feature is useful when the majority of an image requires repeated rendering (excluding camera, light, and other settings). This is the same as rendering a scene twice stereoscopically.

The following functions control the packet recording feature.

**Code 6-1. Functions Used to Control the Packet Recording Feature**

```
nn::gd::System::StartRecordingPackets(
        u32* forceDirtyModuleFlag,
        nn::gd::RecordingPacketUsage usage = RECORD_COMMAND_LIST_COPY);
nn::gd::System::StopRecordingPackets(nn::gd::RecordedPacketId** packetId);
nn::gd::System::ReplayPackets(nn::gd::RecordedPacketId* packetId,
        u32* forceDirtyModuleFlag);
nn::gd::System::ReleasePackets(nn::gd::RecordedPacketId* packetId);
```

You can record both 3D commands output to the 3D command buffer and command requests (see 7.1. Internal Use of nngx Functions) output to the request buffer.

You can use the `nn::gd::System::ReleasePackets()` function to release command packets that you no longer need.

### Effects of the Internal State on Outputting 3D Commands

Depending on the internal state of the GD library, functions do not necessarily output 3D commands. This may cause behavior to deviate from what was expected when the command packets were recorded. Similarly, the state of the GPU may be inconsistent with the internal state of the GD library

when the command packets are replayed. To work around this problem, the `nn::gd::System::StartRecordingPackets()` and `nn::gd::System::ReplayRecordingPackets()` functions each accept flags as arguments, allowing you to indicate which modules to mark as *dirty* before the function is executed.

The safest course of action is to specify the `MODULE_ALL` flag and mark all modules as dirty. This causes all 3D commands to be output and definitely prevents erroneous states. When you use this feature, if you know which module's state will be recorded and replayed and whether that particular state will cause a conflict, you can improve performance by specifying only the minimum number of necessary modules as *`dirty`*.

#### Editing Command Packets

You can edit recorded command packets before you replay them. To calculate the 3D command buffer position to update, combine two values: The first value is the offset for output into the 3D command buffer that is returned by the `nn::gd::System::GetCommandBufferOffset()` function while packets are being recorded (between calls to the `nn::gd::System::StartRecordingPackets()` and `nn::gd::System::StopRecordingPackets()` functions). The second value is the base address of the 3D command buffer that is returned by the `nn::gd::System::GetCommandBufferBaseAddress()` function after you have finished recording packets. However, to overwrite 3D commands, you can only use immediate functions, which output 3D commands to the 3D command buffer.

#### Using the usage Parameter to Issue an Instruction to Copy the 3D Command Buffer

To create a copy of the 3D command buffer, specify `Record_Command_List_Copy` in *`usage`* under the `nn::gd::System::StartRecordingPackets()` function as was done before adding the argument. The 3D command in the command packet that was recorded can be overwritten for this purpose.

On the other hand, specifying `Record_Command_List_No_Copy` in *`usage`* means that a 3D command buffer copy will not be created. This shortens processing time, but the 3D command cannot be overwritten. In addition, clearing the 3D command buffer acting as the source of the copy renders the command packet unexecutable.

When packet recording is started with `RECORD_3D_COMMAND_BUFFER_FOR_JUMP` specified in *`usage`*, the `nn::gd::System::ReplyPackets()` function creates a copy of the 3D command buffer, and it is executed with a jump by inserting a jump command using the `nngxAddSubroutineCommand()` function.



# 7. Relationship Between the GD and GX Libraries

This chapter explains which `nngx()` functions are called by functions in this library, and when you need to pay particular attention to `nngx()` functions.

## 7.1. Internal Use of nngx Functions

Some of the functions in the GD library call nngx() functions. As long as no errors have occurred, the functions in the following table call nngx() functions when the condition in the **Condition** column is met. However, the same function may call different nngx() functions depending on the condition.

If "Always" is given as a condition, the corresponding nngx() functions are always called unless an error has occurred.

**Table 7-1. GD Functions That Call nngx Functions**

| GD Function | Conditions | nngx Functions |
|---|---|---|
| System::<br>StartRecordingPackets | Always | nngxStartCmdlistSave |
| System::<br>StopRecordingPackets | Always | nngxSplitDrawCmdlist<br>nngxStopCmdlistSave<br>nngxExportCmdlist |
| System::ReplayPackets | Always | nngxImportCmdlist |
| System::<br>Execute3DCommandList | Always | nngxSplitDrawCmdlist |
| Resource::<br>CreateTexture2DResource | There is initial data, and data is copied to a region allocated in VRAM. | nngxAddVramDmaCommand |
| | There is initial data, and data is copied to a region allocated in device memory. | nngxUpdateBuffer |
| | There is initial data, and mipmaps are generated automatically. | nngxFilterBlockImage |
| Memory::GenerateMipMaps | Always | nngxFilterBlockImage |
| | Rendering dependency | nngxSplitDrawCmdlist |
| Resource::<br>CreateVertexBufferResource | There is initial data, and data is copied to a region allocated in VRAM. | nngxAddVramDmaCommand |
| | There is initial data, and data is copied to a region allocated in device memory. | nngxUpdateBuffer |
| Memory::<br>CopyTextureSubResource | Always | nngxAddBlockImageCopyCommand |
| | Rendering dependency | nngxSplitDrawCmdlist |
| Memory::<br>CopyVertexBufferSubResource | Always | nngxAddBlockImageCopyCommand |
| Memory::<br>CopyTexture2DResource<br>BlockToLinear | Always | nngxAddB2LTransferCommand |
| | Rendering dependency | nngxSplitDrawCmdlist |
| Memory::<br>CopyTexture2DResource<br>LinearToBlock | Always | nngxAddL2BTransferCommand |
| | Rendering dependency | nngxSplitDrawCmdlist |
| Memory::ClearTargets | Always | nngxAddMemoryFillCommand |
| | Rendering dependency | nngxSplitDrawCmdlist |
| Memory::<br>ClearTexture2DResource | Always | nngxAddMemoryFillCommand |
| | Rendering dependency | nngxSplitDrawCmdlist |
| Resource::<br>UnmapTexture2DResource | A mapping that involves writing to resources in device memory | nngxUpdateBuffer |
| Resource:: | A mapping that involves writing to resources in | nngxUpdateBuffer |

| | | |
|---|---|---|
| UnmapVertexBufferResource | device memory | |

## 7.2. Interaction With nngx Functions

This section explains when you need to pay particular attention to nngx() functions while using the GD library.

### 7.2.1. Creating Textures With Initial Data in VRAM

When you create a resource in VRAM using initial data, a DMA transfer command request is added to the command request queue. This command transfers the initial data from main memory to VRAM. Because the GD library does not manage command request execution, applications must confirm that the DMA transfer has completed before releasing the initial data.

As shown in the following sample code, you can call the nngxWaitCmdlistDone() function to wait for all of the command requests to complete. However, if the command list is double-buffered, the DMA transfer command request is not executed until the next frame after it was added to the current command request queue.

**Code 7-1. Creating Textures With Initial Data in VRAM**

```
nn::gd::Texture2DResource* texture2DResource = 0;
nn::gd::Texture2DResourceDescription Text2DResDesc =
{
    width, height, 1, nn::gd::Resource::NATIVE_FORMAT_RGB_888,
    nn::gd::Memory::LAYOUT_BLOCK_8, nn::gd::Memory::VRAMA
};
nn::gd::Resource::CreateTexture2DResource(
        &Text2DResDesc, data, GD_TRUE, &texture2DResource);

// We need to confirm that the DMA transfer has completed before releasing the
resource, which is created in VRAM.
nngxWaitCmdlistDone();

free(data);
```

### 7.2.2. Accessing Rendered Results

The CPU cannot directly access data in VRAM. If you want to access the contents of the color buffer, depth (stencil) buffer, and textures to directly overwrite something you have already rendered, you must take the following steps.

- Allocate a temporary resource in device memory.
- Copy the resource in VRAM to your temporary resource (transfer data from VRAM).
- Overwrite the temporary resource.
- Write the content of the temporary resource back to the resource in VRAM (transfer data to VRAM).

Command requests transfer data to and from VRAM. You must wait for the command requests to complete before you overwrite the resource or deallocate the temporary resource.

This procedure can be expressed by the following code.

**Code 7-2. Accessing Rendered Results**

```
// Allocate a temporary resource in device memory (FCRAM).
nn::gd::Texture2DResource* tmpResource= 0;
nn::gd::Texture2DResourceDescription tmpResourceDesc =
{
    s_RenderTextureWidth, s_RenderTextureHeight, 1,
    nn::gd::Resource::NATIVE_FORMAT_RGBA_8888,
    nn::gd::Memory::LAYOUT_BLOCK_8,
    nn::gd::Memory::FCRAM
};
res = nn::gd::Resource::CreateTexture2DResource(
        &tmpResourceDesc, 0, GD_FALSE, &tmpResource);

// Copy the resource in VRAM to the temporary resource.
nn::gd::Memory::Rect memRect(0, 0, -1, -1);
res = nn::gd::Memory::CopyTextureSubResource(
        s_texture2DResource_ColorBuffer, 0, memRect, tmpResource, 0, 0, 0);
// Map the resource to access it.
u8* data;
nn::gd::Resource::MapTexture2DResource(
        tmpResource, 0, nn::gd::Resource::MAP_READ_WRITE, (void**)&data);
// You must wait for the command request to complete before accessing data.
nngxWaitCmdlistDone();

// Write to the mapped resource here.
for (int y=0; y<s_RenderTextureHeight*s_RenderTextureWidth>>1; y++)
{
    data[k+] = 255; data[k] = 255; data[k] = 255; data[k+] = 255;
}

// Change to a temporary resource is finalized on unmapping.
nn::gd::Resource::UnmapTexture2DResource(tmpResource);

// Copy the temporary resource to the resource in VRAM.
nn::gd::Memory::CopyTextureSubResource(
        tmpResource, 0, memRect, s_texture2DResource_ColorBuffer, 0, 0, 0);
```



# 8. Coordinating Operations With Other Libraries and Frameworks

This chapter provides cautions for using the GD library along with other graphics-related libraries and frameworks.

## 8.1. Cautions for Initialization

When you use the same command list with other frameworks (such as NintendoWare), make sure that you do not initialize it more than once.

For example, in Code 2-1, only the GD library's initialization function

(`nn::gd::System::Initialize`) is called when NintendoWare's command list-related initialization has finished.

## 8.2. Using nn::gd::System::ForceDirty

The GD library's internal state may lose synchronization with the GPU when some other library or framework has written 3D commands to the 3D command buffer. If this happens, call the `nn::gd::System::ForceDirty()` function to mark several modules as dirty and resend all 3D commands for the corresponding modules.

Modules that have been marked as dirty by the `ForceDirty()` function generate 3D commands when a rendering command is executed. When 3D commands have already been generated, call the `nn::gd::System::FlushDirtyModule()` function.

The GD library does not maintain the content of registers that are set by immediate functions. When you have used an immediate function to alter the content of a register, you must call the immediate function to reset the register, after the 3D commands have been generated.

For more information about the internal state, see 1.2.2. Managing Module States.

## 8.3. Duplicate Command Lists

The GD library has some functions that queue command requests. After command requests are queued in a command list, they are not executed until the command list is executed. You must avoid overwriting or deleting resources handled by those command requests before they are executed.

This caution also applies to frameworks that use two command lists, because command requests are queued in the command list that is used to create the *next* frame.

## 8.4. Inserting Dummy Commands

Dummy commands that are not recognized by the GPU as commands can be added to the command buffer by calling the `nn::gd::System::AddDummyCommands()` function. Dummy commands can be used when you need to wait for the next command to execute, or need to adjust the size of the command buffer.



# 9. Appendix

# 9.1. List of Immediate Functions

The following table lists the immediate functions.

**Table 9.1. Immediate Functions**

| Class | Function |
|---|---|
| nn::gd::CombinerStage | SetTextureCombinerUnitConstantColor |
| nn::gd::FogStage | UploadFogLookUpTableFloat |
| | UploadFogLookUpTableNative |
| | UploadGasLookUpTableFloat |
| | UploadGasLookUpTableNative |
| nn::gd::LightingStage | SetGlobalColorAmbient |
| | UploadGasLookUpTableFloat |
| | UploadLookUpTableNative |
| nn::gd::Light | SetColorAmbient |
| | SetColorDiffuse |
| | SetColorSpecular0 |
| | SetColorSpecular1 |
| | SetPosition |
| | SetDirection |
| | SetSpotDirection |
| | SetLightType |
| | SetDistanceAttenuationScaleBias |
| | EnableTwoSideDiffuse |
| | EnableGeomFactor0 |
| | EnableGeomFactor1 |
| | UploadLookUpTableFloat |
| | UploadLookUpTableNative |
| nn::gd::ProceduralTextureStage | UploadLookUpTableRgbMapFloat |
| | UploadLookUpTableRgbMapNative |
| | UploadLookUpTableAlphaMapFloat |
| | UploadLookUpTableAlphaMapNative |
| | UploadLookUpTableNoiseMapFloat |
| | UploadLookUpTableNoiseMapNative |
| | UploadLookUpTableColorMapFloat |
| | UploadLookUpTableColorMapNative |
| nn::gd::ShaderStage | SetFloatConstantBuffer |
| nn::gd::OutputStage | SetPenumbraScaleBias |
| nn::gd::RasterizerStage | SetCulling |
| | SetViewport |
| | EnableClippingPlane |
| | SetClippingPlane |
| | EnableScissor |

| | |
|---|---|
| | SetScissor |

## 9.2. Memory Allocation Within Functions

The sizes in the following table are passed to the allocator that is specified when the GX library is initialized.

Unless specified otherwise in the **Comments** column, memory is allocated as a system buffer (NN_GX_MEM_SYSTEM) from device memory (NN_GX_MEM_FCRAM).

**Table 9-2. Memory Allocated Within Functions**

| Function | Size | Comments |
|---|---|---|
| System::Initialize | 24 bytes | Management information |
| | 12 bytes | Management information |
| | 12 bytes | Management information |
| | 28 bytes | Management information |
| | 12 bytes | Management information |
| | 1536 bytes | Management information |
| | 52 bytes | Management information |
| | 28 bytes | Management information |
| | 12 bytes | Management information |
| | 1664 bytes | Management information |
| | 56 bytes | Management information |
| | 28 bytes | Management information |
| | 12 bytes | Management information |
| | 2688 bytes | Management information |
| | 28 bytes | Management information |
| | 12 bytes | Management information |
| | 14,208 bytes | Management information |
| | 616 bytes | Management information |
| | 28 bytes | Management information |
| | 12 bytes | Management information |
| | 1664 bytes | Management information |
| | 60 bytes | Management information |
| | 28 bytes | Management information |
| | 12 bytes | Management information |
| | 640 bytes | Management information |
| | 328 bytes | 3D command<br>(Changes the shared processor settings to |

| | | |
|---|---|---|
| | | use the geometry shader.) |
| | 328 bytes | 3D command<br>(Changes the shared processor settings to not use the geometry shader.) |
| Resource::CreateTexture2DResource | 52 bytes | Management information |
| | 16 bytes | Management information |
| | Data-dependent | Allocated with NN_GX_MEM_TEXTURE in the specified memory as long as there is initial data and copies are made. |
| Resource::<br>CreateTexture2DResourceCastFrom | 64 bytes | Management information |
| Resource::Helper::<br>ConvertTextureResourceToNativeFormat | Data-dependent | This memory is allocated for a temporary buffer used by the conversion and is released when the function exits. |
| Resource::Helper::ConvertCompressed<br>TextureResourceToNativeFormat | Data-dependent | This memory is allocated for a temporary buffer used by the conversion and is released when the function exits. |
| Resource::CreateRenderTarget | 48 bytes | Management information |
| Resource::CreateDepthStecilTarget | 52 bytes | Management information |
| TextureStage::CreateTexture2D | 64 bytes | Management information |
| TextureStage::CreateTextureCube | 112 bytes | Management information |
| ShaderStage::CreateShaderBinary | 28 bytes | Management information |
| | Data-dependent | Management information |
| | Data-dependent | Management information |
| | Data-dependent | Management information |
| | Data-dependent | Management information |
| | Data-dependent | Management information |
| | Data-dependent | Management information |
| | 20 bytes | Management information |
| | Data-dependent | 3D commands |
| | Data-dependent | 3D commands |
| ShaderStage::CreateShader | 20 bytes | Management information |
| | Data-dependent | 3D commands |
| ShaderStage::CreateShaderPipeline | 148 bytes | Management information |
| | Data-dependent | 3D commands |
| | Data-dependent | Uniform setting values<br>(vertex shader) |
| | Data-dependent | Uniform setting values (geometry shader) |
| VertexInputStage::CreateInputLayout | 204 bytes | Management information |
| | Data- | InputElementDescription classes |

| | | |
|---|---|---|
| | dependent | |
| Resource::CreateVertexBufferResource | 24 bytes | Management information |
| | Data-dependent | Allocated with NN_GX_MEM_VERTEXBUFFER in the specified memory as long as there is initial data and copies are made. |
| CombinerStage::CreateTextureCombinerState | 616 bytes | Management information |
| | 92 bytes | Management information |
| TextureStage::CreateSamplerState | 60 bytes | Management information |
| OutputStage::CreateDepthStecilState | 52 bytes | Management information |
| OutputStage::CreateBlendState | 56 bytes | Management information |
| System::StartRecordingPackets | 36 bytes | Management information |
| System::StopRecordingPackets | Data-dependent | Recorded 3D commands |

## 9.3. Debugging Support Features

The GD library provides various functions that are useful for debugging, such as filter features that modify the rendering behavior. Filter functions are only enabled when compiled with Debug or Development.

### 9.3.1. Filter Features

Filter features allow the rendering behavior to be modified by setting various flags. Filter features are only enabled when compiled with Debug or Development.

**Code 9-1. Function for Specifying Filter Features**

```
static nnResult nn::gd::System::Debug::SetFilter(
                u32 filterFlag, u32 drawStartIndex, s32 drawCount);
```

The filter features are enabled after the number of rendering operations specified by drawStartIndex, and are enabled for the number of render operations specified by drawCount.

The filters to be enabled are specified in filterFlag by logically ORing flag values from the following table.

**Table 9-3. Filter Feature Flags**

| Flag | Description |
|---|---|
| FILTER_NONE | Specifying this flag by itself disables all filter features. |
| FILTER_TEXTURE_8X8 | Convert all textures used in rendering to 8x8 textures. |
| FILTER_TEXTURE_CUSTOM | Convert all textures used in rendering to the custom textures specified using the SetCustomTexture() function. A fixed 8×8 |

| | |
|---|---|
| | texture is used if no custom textures have been specified. |
| FILTER_SAMPLER_STATE_FILTER_NEAREST_MIPMAP_NEAREST | Forces the filter setting for texture shrinking to GL_NEAREST_MIPMAP_NEAREST. |
| FILTER_SAMPLER_STATE_FILTER_LINEAR_MIPMAP_NEAREST | Forces the filter setting for texture shrinking to GL_LINEAR_MIPMAP_NEAREST. |
| FILTER_VIEWPORT_1X1 | Changes the viewport setting to 1×1. |
| FILTER_SIMPLE_COMBINER | Changes the combiner settings so that all pixels are output as green. |
| FILTER_DISABLE_BLENDING | Disables blending. |
| FILTER_DISABLE_DEPTH_STENCIL_TEST | Disables depth stencil testing. |
| FILTER_DISABLE_ALPHA_TEST | Disables alpha testing. |
| FILTER_LIGHTING_STAGE_1_LIGHT_MAX | Sets the maximum number of light sources to 1. If multiple lights have been configured, only the first light that is enabled will be enabled. |
| FILTER_LIGHTING_STAGE_SIMPLE_LAYER_CONFIGURATION | Changes to the layer configuration requiring the smallest number of cycles. |
| FILTER_DISABLE_DRAW | Disables rendering. |
| FILTER_VISUALIZE_OVERDRAW | Adds a blending configuration so that rendered pixels gradually go white, allowing pixels that are rendered repetitively to be visualized. Using this together with the filter disabling depth stencils allows pixels discarded (or not discarded) in testing to be checked. |
| FILTER_DISABLE_CULLING | Disables culling. |
| FILTER_ENABLE_CULLING_CLOCKWISE | Changes culling to clockwise. |
| FILTER_ENABLE_CULLING_COUNTERCLOCKWISE | Changes culling to counterclockwise. |

## 9.3.2. Mipmap Level Visualization

The `nn::gd::System::Debug::ColorizeMipmaps()` function supports the visualization of mipmap levels by filling in each mipmap level specified in the texture source with a specified color.

**Code 9-2. Function Supporting Visualization of Mipmap Levels**

```
static nnResult nn::gd::System::Debug::ColorizeMipmaps(
                nn::gd::Texture2DResource* texture2DResource,
                s32 minMipLevelIndex, s32 maxMipLevelIndex, u8 colors[][4]);
```

The minimum and maximum indices of the mipmap levels to be filled in are specified in `minMipLevelIndex` and `maxMipLevelIndex` respectively. When −1 is specified for either of these parameters, the maximum mipmap level of the texture resource specified by `texture2DResource` is specified.

The colors specified in `colors` are used for the mipmap levels, starting at `minMipLevelIndex`. In other words, the texture of the mipmap level specified by `minMipLevelIndex` is filled in with the color specified by `colors[0]`, and the texture of the next mipmap level is filled in with the color specified by `colors[1]`.

If NULL is specified for `colors`, a default color array is used to fill in the textures. For information about the default color array, see the API Reference.

# Revision History

## Version 1.1 (2016-05-10)

### Changes

- 1.1.1. Command Lists
    - Added that the function issuing command requests can only be called in Core 0.

## Version 1.0 (2014-09-04)

### Additions and Changes

- Initial version.

# 3DS Programming Manual: Wireless Communication

Version 1.6

**Nintendo Confidential**

This document contains confidential and proprietary information of Nintendo, and is protected under confidentiality agreements as well as the intellectual property laws of the United States and of other countries. No part of this document may be released, distributed, transmitted, or reproduced in any form , including by any electronic or mechanical means and by including within information storage and retrieval systems, without written permission from Nintendo.

---

©2015–2016 Nintendo. All rights reserved.

All company and product names in this document are the trademarks or registered trademarks of their respective companies.

CTR-06-0012-002-F

# 1. Introduction

This document provides an overview of the various libraries used with the wireless communication module installed in the 3DS system and describes functions and programming procedures.

2. Overview of Wireless Communication provides an overview of wireless communication and includes introductions to the types of wireless communication and the relevant libraries that use the wireless communication module.

3. Foreground Communication outlines the libraries related to wireless communication that are intentionally run by the application, called *foreground communication*.

4. Background Communication outlines the libraries related to wireless communication being performed in the background, called *background communication*.

5. Supplemental Communication explains the libraries for assisting applications using wireless

communications, such as the blocked-user list intended to prevent the spread of inappropriate user-generated content.

6. Debugging Libraries explains how to use wireless communication libraries that have been provided for debugging but cannot be used in retail products.



# 2. Overview of Wireless Communication

This chapter describes the types of wireless communication that can be performed by applications, and the system's wireless-enabled and wireless-disabled modes.

## 2.1. Types of Wireless Communication

3DS wireless communication is mainly split into two types: *foreground communication*, which is carried out by the application intentionally; and *background communication*, which is performed in the background.

Multiple libraries are provided for both foreground and background communication, depending on the intended use.

**Table 2-1. Wireless Communication Categorized as Foreground Communication**

| Type of Wireless Communication | Library Name | Namespace | Description |
|---|---|---|---|
| Local Communication (UDS Communication) | UDS | `nn::uds` | Direct wireless communication between 3DS systems.<br>Up to 16 (TBD) systems can connect to a single network.<br>Broadcasts are also supported. |
| Nintendo 3DS Download Play | DLP | `nn::dlp` | Wireless communication for distributing child communication programs.<br>You can distribute programs up to 32 MB in size. |
| Automatic Connection | AC | `nn::ac` | A library that automatically establishes an Internet connection, through a wireless access point or otherwise, in accordance with System Settings. |
| Reliable local communication (RDT) | RDT | `nn::rdt` | Wireless communication for reliable one-to-one data transmission.<br>Uses UDS communication internally. |

**Note:** Nintendo supports 16 simultaneous UDS communication connections, but operation has

not been adequately tested when 13 or more systems are simultaneously connected. Consequently, proper operation with 13 or more connected systems is not currently guaranteed.

**Table 2-2. Wireless Communication Categorized as Background Communication**

| Type of Wireless Communication | Library Name | Namespace | Description |
|---|---|---|---|
| StreetPass | CEC | `nn::cec` | Wireless communications for sending and receiving data when two CTR systems approach and discover each other as communication peers. |
| SpotPass | BOSS | `nn::boss` | Wireless communications that use SpotPass functionality. SpotPass executes tasks registered by an application in the background. |
| Presence feature | FRIENDS | `nn::friends` | Feature for setting the presence information reported to friends and for getting the presence information about friends. |

## 2.2. Wireless-Enabled and Wireless-Disabled Modes

The wireless switch incorporated into the CTR and SPR systems provides hardware controls that use the wireless communication module. The SNAKE and FTR systems, which do not include a physical wireless switch, have a wireless switch on the HOME Menu that fulfills the same role.

The term *wireless-disabled mode* refers to the state where the wireless communication module has been turned off using the wireless switch.

Wireless signals are not actually sent and received even in wireless-enabled mode unless communications are being performed by the wireless module in the background or by the UDS library.

If the system is in wireless-disabled mode, wireless signals are never sent or received.

The yellow LED indicator for wireless operations turns on and off according to the status of the wireless communication module.

**Table 2-3. Correspondence Between Wireless Communication Module Status and Wireless LED Status**

| Wireless Communication Module Status | Wireless LED Status |
|---|---|
| Wireless-enabled mode | Lit |
| Wireless off mode | Off |
| Transmitting wireless data | Flashing |

No functions have been prepared for checking wireless-enabled mode and wireless-disabled mode. When in wireless-disabled mode, the library's functions for initialization and for wireless sending and receiving return values that indicate it is in wireless-disabled mode.

## 2.3. Processing Load of Encrypted Wireless Communication

Encryption is necessary for communication with access points configured with WEP or WPA authentication, and the wireless communication module performs this encryption in the hardware. Although the CPU performs some of the processing for WPA connections, this processing takes place in the system core.

The application does not need to take the processing load of encrypted wireless communication into account.

## 2.4. Errors From the Wireless Communication Module

Due to differences between individual 3DS systems, there is a small chance that errors occur between the CPU and the wireless communications module. This type of error cannot be resolved through software revisions. As a workaround, we have implemented automatic recovery when such an error occurs.

However, one of the errors that can occur due to individual differences in systems cannot be handled by the automatic recovery. It is RSL:F9606C02 (RSL is a result code displayed on a FATAL screen), and occurs when local communication or an automatic connection is being initialized or the communication mode of background communication is changing and the process is terminated. When this error occurs, a FATAL screen is displayed, so there is no need for the application to do anything.

> **Note:** If the RSL:F9606C02 error occurs frequently when using development tools or a test unit, it may be because the wireless communication module is damaged, and not due to individual system differences. Please consult with Nintendo if this is the case.

When a wireless communication module error occurs, the behavior can differ as described below, according to the communication mode.

- Local Communication (UDS, RDT, or Nintendo 3DS Download Play)
  Automatic recovery works. As a result, if an API is called and this error occurs, values such as `nn::uds::ResultWirelessOff` or `nn::dlp::ResultWirelessOff` are returned, indicating that wireless communication is in off mode.

- Automatic Connection (AC)
  Automatic recovery works. As a result, if an AC API is called and this error occurs, the value `nn::ac::ResultWifiOff` is returned, indicating that wireless communication is in off mode.
- Background Communication (system is open)
  Automatic recovery works in any communication mode (StreetPass, SpotPass, Friend Presence). The application does not need to do anything.
- Background Communication (system is closed)
  Automatic recovery is not performed, and system power is turned off. The application does not need to do anything.
- Wireless-disabled mode
  The communication error will not occur.

The following is a list of FATAL errors that could occur for each communication mode.

**Table 2-4. Fatal Errors by Communication Mode**

| Communication Mode | Result Code and Frequency |
|---|---|
| Local communication | RSL:F9606C02<br>Occurrence Frequency: Very low |
| Automatic Connection | RSL:F9606C02<br>Occurrence Frequency: Very low |
| Background Communication (system is open) | RSL:F9606C02<br>Occurrence Frequency: Very low |
| Background Communication (system is closed) | Power is turned off without displaying a FATAL screen.<br>Occurrence Frequency: Very low |
| Wireless off mode | Does not occur. |

CONFIDENTIAL

# 3. Foreground Communication

The following libraries have been provided for applications to intentionally use the wireless communication module for wireless communication.

- Local Communication (UDS Communication)
- Nintendo 3DS Download Play
- Automatic Connection
- Reliable Local (RDT) Communication

This chapter describes programming procedures and other information necessary to develop applications using each of these libraries.

# 3.1. Nintendo 3DS Local Play (UDS) Communication

The 3DS platform uses *Nintendo 3DS Local Play (UDS) communication* as the Nintendo-specific local communication method intended for use in games.

Nintendo 3DS Local Play (UDS) communication has the following characteristics.

- Allows up to 16 systems to be simultaneously connected to a single network (although proper operation with 13 or more connected systems is not guaranteed, at present).
- Supports packet broadcasting.
- Not synchronized with picture frames.
- Does not guarantee that communication data will arrive.
- Encrypts communication channels.
- Preserves connections even if data is not sent or received regularly.

**Warning:** Nintendo 3DS Local Play (UDS) communication is not available between development CTR systems (debugger and development system) and retail CTR systems.

Nintendo 3DS Local Play (UDS) communication uses the UDS library provided by the CTR-SDK.

The following terms are used with the UDS library.

**Node**

A terminal (device) that sends and receives data on a UDS communication network.

Hosts and clients (described later) are both types of nodes.

**Node ID**

An ID that is assigned when a node connects to a UDS communication network.

For more information, see 3.1.8. Node ID Assignment.

**Host**

A node that establishes a new UDS communication network and can change network attributes as necessary.

Corresponds to the "MP Parent" in MP communications for the Nintendo DS system or Nintendo DSi system.

#### Client

A node that sends and receives data after connecting to an existing network established by a host.

Corresponds to the "MP Client" in MP communications for the Nintendo DS system or Nintendo DSi system.

#### Spectator

A system, but not a node, that only receives data after connecting to an existing network established by a host.

Unlike a client node, a spectator system does not count toward the maximum number of simultaneous network connections. The library does not place any limitations on the number of simultaneous spectator connections. The application also cannot limit the number of connections.

#### Unicast communication

Unicast communication is a method to send packets from a node to one other node, and is typically between a host and a client. Client-to-client unicast communication is possible, but two transmissions are performed within the library because the communication must pass through the host. The extra transmission increases the latency of data sent between clients, but it also ensures that the recipient node is never out of communication range.

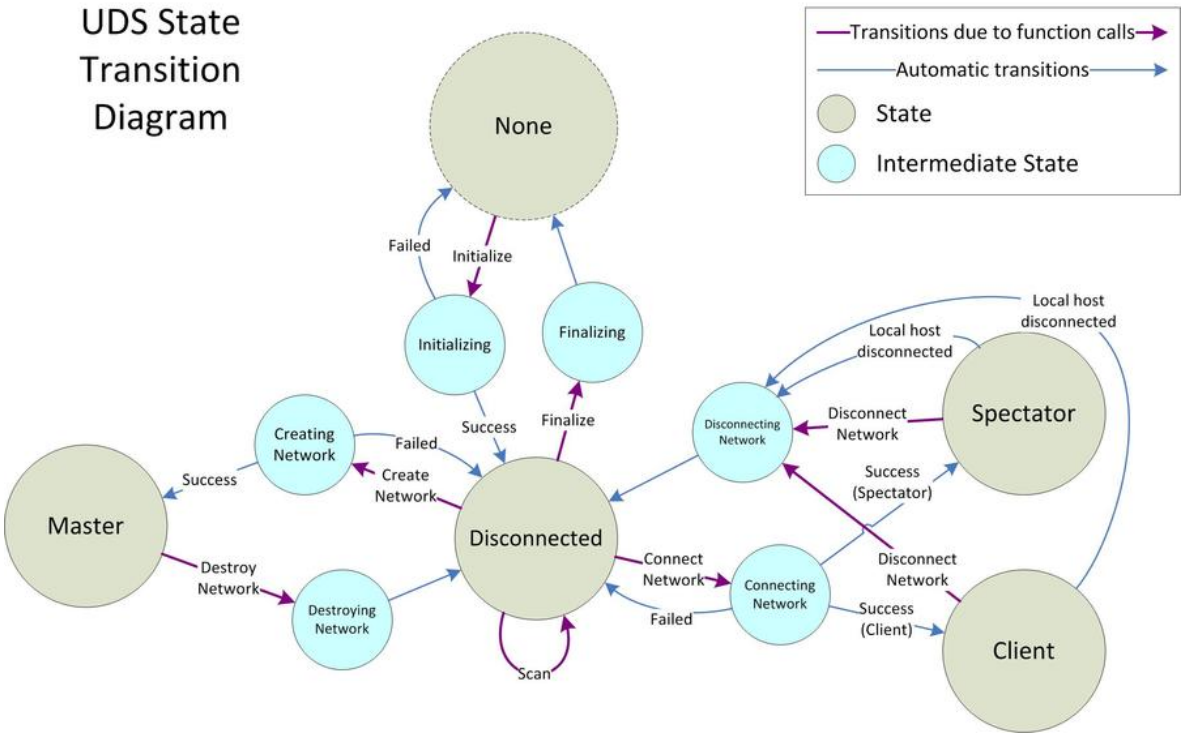
#### Broadcast communication

Broadcast communication is a method to send packets from a node to all other nodes. It enables packets to be sent to multiple nodes via a single transmission. Communication via this method is processed as a single transmission, regardless of how many nodes there are in the network. Although this reduces the latency of data sent from clients, note that transmissions will not reach clients that are out of communication range.

## 3.1.1. Internal UDS Communication States

Figure 3-1 shows internal state transitions for the UDS library.

**Figure 3-1. State Transition Diagram**




There are two types of internal states: *states*, which show the current status, and *intermediate states*, through which the current state transitions to another. Intermediate states exist for internal library processing. Although application developers generally do not need to pay attention to them, if the current state is an intermediate state, wait for it to transition to a normal state.

The UDS library restricts functions that can be called from some states. An error is returned immediately if you call a function from a state in which it is unusable.

**Table 3-1. States**

| State Name | Description |
|---|---|
| None | Has not yet initialized the UDS library. |
| Disconnected | Has initialized the UDS library but does not belong to a network. |
| Host | Created a new network and is running as a host node. |
| Client | Connected to an existing network as a client node. |
| Spectator | Connected to an existing network as a spectator node. |

**Table 3-2. Intermediate States**

| Intermediate State Name | Description |
|---|---|
| Initializing | Initializing the UDS library. When it is successful, this state transitions to `Disconnected`. |
| Finalizing | Shutting down the UDS library. After it is finished, this state transitions to None. |
| Creating Network | Creating a network based on the settings. After it is finished, this state transitions to `Master`. |
| Connecting | Connecting to an existing network as a client or spectator node. |

| | |
|---|---|
| Network | When it is successful, this state transitions to `Client` or `Spectator`. |
| Destroying Network | Disconnects all client and spectator nodes from the network and then destroys the network.<br>After it is finished, this state transitions to `Disconnected`. |
| Disconnecting Network | Leaving the current network connection. After it is finished, this state transitions to `Disconnected`.<br>A node also enters this state when it is disconnected from the network because of an external cause. |

These states are defined by `nn::uds::State`. The following table shows how the names of the states and definitions correspond.

**Table 3-3. State Names and Their Corresponding Definitions**

| State Name | Definition Name |
|---|---|
| None | `STATE_NONE` |
| Disconnected | `STATE_DISCONNECTED` |
| Host | `STATE_MASTER` |
| Client | `STATE_CLIENT` |
| Spectator | `STATE_SPECTATOR` |
| Initializing | `STATE_PROCESSING_INITIALIZE`<br>(The application never gets this state.) |
| Finalizing | `STATE_PROCESSING_FINALIZE`<br>(The application never gets this state.) |
| Creating network | `STATE_CREATING_NETWORK` |
| Connecting to network | `STATE_CONNECTING_NETWORK` |
| Destroying the network | `STATE_DESTROYING_NETWORK` |
| Disconnecting from the network | `STATE_DISCONNECTING_NETWORK` |

## 3.1.2. Initializing

Call `nn::uds::Initialize()` to initialize the UDS library.

**Code 3-1. Initializing the UDS Library**

```
nn::Result nn::uds::Initialize(
        nn::os::Event* pStatusUpdateEvent,
        void* receiveBuffer, const size_t bufferSize);
nn::Result nn::uds::Initialize(
        nn::os::Event* pStatusUpdateEvent,
void* receiveBuffer, const size_t bufferSize,
        nn::cfg::UserName* pUserName);
```

Set *pStatusUpdateEvent* equal to a pointer to an instance of the `nn::os::Event` class. This instance is initialized as an auto-reset event.

The `Event` instance is initialized within `nn::uds::Initialize()` and is used to notify the application at the following times.

- When the calling node creates or connects to a network.
- When the calling node disconnects or is disconnected from the network.
- When another node connects to the network.
- When another node disconnects or is disconnected from the network.

Set *receiveBuffer* and *bufferSize* to the buffer for receiving data used within the UDS library and its size, respectively. The application must **allocate this memory from non-device memory** with the starting address `nn::uds::BUFFER_ALIGNMENT` (4096 bytes) aligned, and the size must be a multiple of `nn::uds::BUFFER_UNITSIZE` (4096). Do not allow applications to access this memory until `nn::uds::Finalize()` has completed finalizing the memory specified for the receive data buffer.

Specify *pUserName* to use a user name that is different from the name in System Settings (for example, when naming a Mii). Handle user names according to the *CTR Guidelines: UGC*. Specify NULL to use the user name in System Settings. This is the same as initializing the library with the version of the function that does not have a *pUserName* parameter.

If `nn::uds::resultAlreadyOccupiedWirelessDevice` is returned in the return value, UDS communication cannot be started because the wireless communication module is being used for other communications. `nn::uds::ResultWirelessOff` is returned if the system is in wireless-disabled mode.

## 3.1.3. Connecting

A UDS communication network is created after the UDS library has been initialized. The following sections describe the procedures for creating a network as a host node and connecting to a network as a client node or spectator system.

### 3.1.3.1. Generating Local Communication IDs

You must first specify a 32-bit local communication ID to identify networks before calling functions to create or search a UDS communication network. Use the following function to generate a local communication ID by converting from a 20-bit unique ID value.

**Code 3-2. Generating Local Communication IDs**

```
bit32 nn::uds::CreateLocalCommunicationId(bit32 uniqueId,
```

```
                                        bool isDemo = false);
```

For *uniqueId*, specify the unique ID assigned by Nintendo Licensing Department to each title. When communicating between different titles, specify the unique ID for either one of them. An ID value of 0 is reserved by the library and cannot be used.

If you want to disable UDS communication between retail versions and download application demo versions that have the same unique ID, set *isDemo* to `true` on the demo version only. Always set this parameter to `false` in the retail version, regardless of whether you want to communicate with demo versions.

> **Note:** For test programs and titles that use UDS communication and have not been allocated a unique ID by Nintendo, specify a prototype software code in the range `0xFF000` to `0xFF3FF` for *uniqueId*. In retail software, you must always specify the unique ID assigned by Nintendo.

## 3.1.3.2. Creating a New Network

Build a UDS communication network by calling `nn::uds::CreateNetwork()` on the host.

**Code 3-3. Creating a Network**

```
nn::Result nn::uds::CreateNetwork(
        u8 subId,
        u8 maxEntry,
        bit32 localId,
        const char passphrase[], size_t passphraseLength,
        u8 channel);
nn::Result nn::uds::CreateNetwork(
        u8 subId,
        u8 maxEntry,
        bit32 localId,
        const char passphrase[], size_t passphraseLength,
        u8 channel,
        const void* pData, size_t dataSize);
```

The *subId* parameter is the communication mode ID. The application can set this freely. This is expected to be used to distinguish between communication modes such as "network battle" and "data exchange." Do not specify `0xFF` for *subId*. The client uses the value `0xFF` to search the network for all *subId*'s.

For *maxEntry*, specify the maximum number of nodes that can connect to the network. You can specify up to 16 nodes including the host and clients, but operation has not been adequately tested with 13 or more simultaneously connected systems. Consequently, proper operation with 13 or more connected systems is currently not guaranteed. Note that the host itself is included in the node count, but spectator systems are not.

For *localId*, specify the local communication ID generated from the unique ID.

Set *passphrase* to a string denoting the type of encryption key used to encrypt the wireless layer. Communication is established when a combination of *localId* and *passphrase* matches between the host and the client or spectator. Avoid using passphrases that are easily determined, such as extremely short strings, strings that are easy to guess, and strings that you have specified to third parties (including strings used in other titles).

Set *passphraseLength* to the length of the *passphrase*. You can specify a value in the range from 8 to 255 (inclusive).

For *channel*, specify the channel to use for UDS communication: 0, 1, 6, or 11. If 0 is specified, the channel is selected automatically. This channel specification is used only for debugging.

Specify the data and its byte size in *pData* and *dataSize*, when setting optional data in a beacon at the same time as the build. When `0` is specified for *dataSize*, an error will not occur when *pData* is NULL. For more information about setting data in a beacon, see 3.1.3.5. Setting Optional Data in a Beacon.

**Warning:** The channel is always selected automatically when executed on production version hardware.

To build a network that uses a specified channel, you must first set the system to debugging mode. (Using the Config tool, go to **Debug Settings** and set **Debug Mode** to **enable**.)

With UDS communication, a `TemporaryID` is generated randomly to identify each time the network is rebuilt. However, when the time between rebuilding is short, and there were no connections from clients and no communication with other devices, the `TemporaryID` is not updated.

## 3.1.3.3. Finding Nearby Networks

Clients and spectators must search for a nearby network built with the same local communication ID by calling `nn::uds::Scan()`.

**Code 3-4. Searching for an Existing Network**

```
nn::Result nn::uds::Scan(
        void * pBuffer, size_t bufferSize,
        u8 subId,
        bit32 localId);
```

For *pBuffer*, specify the address of the buffer in which to store network information found by the search. Each network requires (approximately) a 1-KB buffer. Network information is not saved beyond the buffer size.

*subId* is the communication mode ID to search. The network to search is the one formed with the same value as specified in *subId*. Specify a value of `0xFF` to search all networks.

For *localId*, specify the local communication ID generated from the unique ID.

> **Warning:** Development CTR systems (debugger and development system) and retail CTR systems cannot discover each other through scanning.

Next, the buffer obtained by running `nn::uds::Scan()` is passed to a `nn::uds::ScanResultReader` constructor to generate an instance of that class for analyzing the search results.

You can call the `GetFirstDescription()` member function on the generated class instance to get the first search result. You can use the `GetNextDescription()` function to get the next search result. Search results are stored in the order of strongest received signal strength indicator (RSSI value).

The search results are given as an instance of the `nn::uds::NetworkDescriptionReader` class. You can use this instance's member functions to analyze network information. The `GetNetworkDescription()`, `GetNodeInformationList()`, and `GetRadioStrength()` functions get network information, a list of nodes, and the signal strength (as one of four levels), respectively. You can also use `CompareWith()` to compare network information. One of three results is returned: "Exactly the same," "Same network but different data," or "Different network."

## 3.1.3.4. Connecting to a Network

A client or spectator can get a variety of information by analyzing search results. Of this, the network information obtained by `GetNetworkDescription()` can be used to call `nn:uds::ConnectNetwork()` and connect to an existing network.

**Code 3-5. Connecting to a Network**

```
nn::Result nn::uds::ConnectNetwork(
        const NetworkDescription & networkDescription,
        ConnectType type,
        const char passphrase[], size_t passphraseLength);
```

For *networkDescription*, specify the network information obtained by a network search.

For *type*, specify the type of the connecting node. You can select one of the following two node types.

**Table 3-4. Network Connection Nodes**

| Setting Value | Description |
|---|---|
| `CONNECT_AS_CLIENT` | Connect to the network as a Client node, which can send and receive data and is given a node ID when it connects to the network. |
| | Connect to the network as a Spectator node, which can only receive |

| | |
|---|---|
| CONNECT_AS_SPECTATOR | data and is not given a node ID when it connects to the network. |

Check whether a system can connect to a network as a spectator by calling the `nn::uds::NetworkDescription` network information class's `CanConnectAsSpectator()` member function and looking for a return value of `true`.

Set *passphrase* to a string denoting the type of encryption key used to encrypt the wireless layer. Communication is established if the combination of the passphrase and the local communication ID of the network built by the host match.

Set *passphraseLength* to the length of the *passphrase*. You can specify a value in the range from 8 to 255 (inclusive).

**Note:** The function returns a `ResultAlreadyNetworkIsFull` error when connected to a network that is already full, and a `ResultNotFoundNetwork` error when the connected network no longer exists.

## 3.1.3.5. Setting Optional Data in a Beacon

The host that established the network can set up to `NET_DESC_APPDATA_SIZE_MAX` bytes of optional data in the network beacon frame.

**Code 3-6. Setting Optional Data in a Beacon**

```
nn::Result nn::uds::SetApplicationDataToBeacon(
                        const void* pData, size_t dataSize);
```

*pData* and *dataSize* specify the starting address of the optional data and the size.

The optional data set in the beacon may include such information as the session state (for example, waiting for opponents or in a match) or a comment. This feature was designed to enable applications to manage their own connections, such as informing clients and spectators who found the network of the network status. If you reset this optional data, however, it is not possible to notify clients and spectators that are already connected of these changes.

It is important to note that optional data is not encrypted. Beacons are easily intercepted by PCs and other devices. Although it is possible to detect if a beacon has been tampered with, it is not possible to detect if devices are using an unmodified beacon that has been intercepted. To prevent the use of specific data in a beacon, you must employ countermeasures to ensure that the data cannot be interpreted if it is intercepted and redistributed by a spectator.

## 3.1.3.6. Getting Optional Data from a Beacon

Client and spectator systems may get the optional data set in the beacon frame by the host by calling member functions for the network information obtained after a network search, or by calling functions for getting information while connected to a network.

**Code 3-7. Getting Optional Data From a Beacon**

```
size_t nn::uds::NetworkDescription::GetApplicationData(
                          bit8* buffer, const size_t bufferSize) const;
nn::Result nn::uds::GetApplicationDataFromBeacon(
                          void* pBuffer, size_t* pDataSize, size_t bufferSize);
```

*buffer*, *bufferSize*, *pBuffer*, and *bufferSize* specify the starting address of the buffer storing the optional data and the buffer size. Be sure to specify the maximum optional data size for the optional data (NET_DESC_APPDATA_SIZE_MAX) and allocate a buffer of that size.

You can check the size required for the obtained optional data from the `nn::uds::NetworkDescription::GetApplicationData()` function return value, and from the *pDataSize* parameter to the `nn::uds::GetApplicationDataFromBeacon()` function.

## 3.1.4. Communicating

At this point, preparations to send and receive data via UDS communication should be complete. The following sections describe how to send and receive data.

### 3.1.4.1. Creating an Endpoint

Before you send or receive data, you must create a network endpoint. An endpoint is a gateway for sending and receiving data and is created with `nn::uds::CreateEndpoint()`.

> **Note:** You can currently create up to 16 endpoints at the same time.

We recommend creating separate endpoints for sending and receiving. The same endpoint can be used for both sending and receiving, but be careful to avoid using an endpoint currently in the midst of sending or receiving for a separate send or receive transaction.

To create an endpoint for receiving data, call `nn::uds::Attach()` and associate the endpoint with the port number and node ID. When you do so, the library creates a reception buffer in the memory block allocated via `nn::uds::Initialize()`.

**Code 3-8. Associating an Endpoint With a Port Number and Node ID**

```
nn::Result nn::uds::Attach(
    EndpointDescriptor * pEndpointDesc,
```

```
    u16 srcNodeId,
    u8 port,
    size_t receiveBufferSize = ATTACH_BUFFER_SIZE_DEFAULT);
```

In *pEndpointDesc*, specify the endpoint you created with nn::uds::CreateEndpoint().

Set *srcNodeId* to the node ID of the sender. Only data sent from the specified node is stored in the reception buffer. Specify BROADCAST_NODE_ID to store data sent from all nodes in the reception buffer.

Set *port* to the number of the port on which to receive data. Only data sent to the specified port number is stored in the reception buffer. Port number 0 is reserved for the system and cannot be used.

Set *receiveBufferSize* to the size of the allocated reception buffer. The minimum value allowed for this parameter is ATTACH_BUFFER_SIZE_MIN. Note that the total size of all reception buffers must not exceed the size of the memory block allocated by nn::uds::Initialize() because the reception buffers are allocated from this memory block. Memory is allocated in units of 32 bytes, so you can make the most efficient use of memory by allocating sizes that are multiples of 32. The default allocates an area the size of ATTACH_BUFFER_SIZE_DEFAULT (eight times UDS_PACKET_PAYLOAD_MAX_SIZE) as the reception buffer.

You can also get the node ID of the sender when receiving data. This allows you to identify the sender even if you receive data sent to the same port by multiple nodes using a single endpoint. The reception buffer is a ring buffer, however, which means that it is possible to lose the earliest packets received if your buffer is not large enough.

When you no longer need to conduct communication through the endpoint, such as when disconnecting from the network, free the endpoint by using nn::uds::DestroyEndpoint(). When you destroy a reception endpoint, the library frees the reception buffer created by the call to nn::uds::Attach().

## 3.1.4.2. Sending

Data is sent to a node by nn::uds::SendTo() through the created endpoint. Note that the function blocks until sending is complete.

**Code 3-9. Sending Data**

```
nn::Result nn::uds::SendTo(
        const EndpointDescriptor & endpointDesc,
        const void * data, size_t dataSize,
        u16 destNodeId, u8 port,
        bit8 option = 0x00);
```

For *endpointDesc*, specify the endpoint that you created.

For *data*, specify a pointer to the data to send. For *dataSize*, specify the size of that data. The maximum amount of data that can be sent by a single function call is defined by UDS_PACKET_PAYLOAD_MAX_SIZE. Setting *dataSize* to a value greater than the maximum size

causes a ResultTooLarge error to be returned. The function returns a ResultMisalignedAddress error if the starting address of the data to send is not 4-byte aligned.

For *destNodeId*, specify the node ID where data is to be sent. Specify a value of BROADCAST_NODE_ID (0xFFFF) to broadcast the data. A spectator can only receive data that is broadcast via this method. It is not possible to send data to a specific spectator. Spectators also cannot send data.

For *port*, specify the port number. Port number 0 is reserved for the system and cannot be used.

Specify the bitwise OR of the following flags for *option*. The specification for this argument can be omitted by setting 0x00. If the option setting is not used, unicast communications from client to client pass through the host. In addition, with the exception of packets passed through the host, packet buffering is enabled by the send buffer.

- NO_WAIT
  If this flag is specified, data is sent immediately without copying it to the UDS send buffer, but execution is blocked until sending is complete.
  If this flag is not specified, the library keeps the transmit buffer small to prevent transmission delay as much as possible, allocating the amount that can be sent at a time (about the same as UDS_PACKET_PAYLOAD_MAX_SIZE). It then sends the data in the buffer when the buffer becomes full, or when a set time (the maximum transmission delay time) has elapsed since the earliest data in the buffer was stored. Because of this, when this flag is not set and data is sent repeatedly, there are cases when data is just accumulated in the buffer, and others when the buffer becomes full and the transmission process is triggered. Blocking time is longer for the latter than for the former.
  The default maximum transmission delay time is set to 10 ms, but it can be set within the range from 5 to 100 ms using nn::uds::SetMaxSendDelay().

> **Note:** The decision about whether to specify NO_WAIT depends on the real-time responsiveness required by a particular communication.
>
> Specify NO_WAIT if you want to minimize the time it takes data to reach the receiver and shorten the time required to exchange data as much as possible.
>
> If Internet latencies are not a problem for your application, you can handle more packets by not specifying NO_WAIT. This improves communication bandwidth when sending a large volume of small packets, because packets are gathered together and then sent. Although throughput does not improve, the number of packets that can be sent per second increases.

- FORCE_DIRECT_BC
  Direct broadcast communication is used to send data for any receiver. Unicast communication between clients does not pass through the host. Although this decreases transmission latency, senders must take into account nodes outside their communication range.

### Transmission Sequence of UDS Communication

The library accumulates packets processed via SendTo() in a radio frame (unit of communication under wireless standards). If the NO_WAIT option is not specified, it sends the packets after accumulating as much data as possible in the send buffer. If the NO_WAIT option is specified, it sends the packets immediately.

When a node receives a radio frame, the library breaks it up into packets and assigns each to a reception buffer. At this point, nodes other than the host discard all packets not addressed to them. UDS communication (which is a star network) is designed so that the host accumulates packets addressed to other nodes in its send buffer and resends them, to enable communication between clients.

#### Guaranteed Order of Sent Packets

When data is sent to the same receiver using the same option settings (FORCE_DIRECT_BC), the order of the packets is guaranteed.

If you do not specify an option, or specify options differently, the order of packets sent via unicast and broadcast communication may change due to differences in the paths they traverse. Nevertheless, because changing the order of packets sent via unicast and broadcast communication is not likely to cause problems in and of itself, you can essentially assume that the order of packets is guaranteed.

#### Send Delay

The library is designed so that small packets are gathered together and then sent to increase the efficiency of the physical layer of wireless communication. When data is sent using nn::uds::SendTo(), there is a chance of waiting to send data for a set send delay time. To avoid this send delay, include NO_WAIT in the *option* parameter specified to nn::uds::SendTo(), when you want to prioritize latency over line efficiency.

Use nn::uds::SetMaxSendDelay() to specify the maximum send delay time.

**Code 3-10. Maximum Send Delay Time Setting**

```
nn::Result nn::uds::SetMaxSendDelay(nn::fnd::TimeSpan maxDelay);
```

Specify a value in the range from 5 to 100 ms for the maximum send delay time of *maxDelay*. The default is 10 ms.

This function can only be called when the system is not connected to the network.

## 3.1.4.3. Receiving

The library receives data sent via the endpoint you associated with the port number and node ID

by the call to `Attach()`. Call `nn::uds::Receive()` if you do not need to get the node ID of the sender, or `nn::uds::ReceiveFrom()` if you do need to get the node ID of the sender. This section describes the `nn::uds::ReceiveFrom()` function.

**Code 3-11. Receiving Data**

```
nn::Result nn::uds::ReceiveFrom(
        const EndpointDescriptor & endpointDesc,
        void * pBuffer,
        size_t * pReceivedSize,
        u16 * pSrcNodeId,
        size_t bufferSize,
        bit8 option = 0x00);
```

For *`pEndpointDesc`*, specify an endpoint created by `nn::uds::CreateEndpoint()` and attached by `nn::uds::Attach()`.

For *`pBuffer`*, specify a pointer to the address used to store received data. The buffer starting address and size must both be 4-byte aligned. The function returns a `ResultMisalignedAddress` error or a `ResultMisalignedSize` error if the starting address or size of the receive buffer is not 4-byte aligned.

For *`pReceivedSize`*, specify a pointer to the address used to store the size of the received data. The maximum amount of data that can be received by a single function call is defined by `UDS_PACKET_PAYLOAD_MAX_SIZE`.

For *`pSrcNodeId`*, specify a pointer to the address used to store the sender's node ID.

For *`bufferSize`*, specify the size of the buffer used to store the received data.

Specify `NO_WAIT` for *`option`* if the function is to end immediately, even if no data has been received. Omit *`option`* for the function to continue until either data is received or an error occurs.

#### The Possibility of Receiving Corrupted Packets

The hardware performs a check for corrupted data when UDS communications are used, but because the emphasis is on speed and adaptability, no checks are conducted at the software level for the completeness of individual packets.

If corruption or tampering will be a problem for your data (for example, when exchanging items), have your application conduct the check at the level of individual files or whatever level is appropriate. There is no need to check at the level of individual packets if tampering will not be a problem, or if your application requires real-time responsiveness.

## 3.1.4.4. Causes of Packet Loss

This section describes the causes of packet loss in UDS communication.

### Degradation of the Environment

Degradation of the communication environment increases packet loss. Examples of such degradation include loss of signal strength due to the distance between the systems, interference due to radio transmissions by third parties, and noise and other external factors.

Actual measurements of packet loss (the library does not guarantee these values) range from about 1% (under good conditions) up to about 10% (when conditions have degraded to the limit of ability to maintain a connection). Nintendo engineers have noted a tendency for packet loss to rise very suddenly and steeply when the system or device cannot keep up with the processing load.

### Connection Speed

As a rule of thumb, the maximum communication volume for UDS is about 500 to 600 packets per second over the entire network. You may experience packet loss if this volume is greatly exceeded or if neighboring networks put stress on your communication bandwidth.

In this case, the rate of packet loss depends on the volume of packets being sent. You can lower this rate by reducing the number of systems that are connected.

Nintendo has confirmed an increase in packet loss in the `com_demo1` demo when a simple communication program communicating on a single network exceeds around 800 packets per second.

### Reception Buffer Overflows

If your application receives packets at a faster pace than it calls the receive function, the reception buffer could overflow and the earliest packets could be dropped. If this happens, the rate of lost packets gradually increases because of the increase in the communication load. Although you can handle temporary increases in load by creating a larger reception buffer, this technique does not remedy chronic buffer overflows.

> **Warning:** The difference in processing speed on SNAKE and CTR causes communication problems. Implement your communication code in a manner that is not overly dependent on the speed of the processor.
>
> For example, use a timer to delay processing after sending packets to avoid sending too many packets or have the sender check the receive status of the receiver as necessary.

## 3.1.4.5. Communicating More Efficiently

UDS communication is based on the IEEE 802.11b/g standard. Under this standard, the entire

network on a channel shares radio bandwidth. The standard also has *backoff* periods of non-communication to avoid interference with the transmission of radio frames. You can improve efficiency by reducing the number of radio frames that are sent, even if the size of the data you send is the same, by such means as accumulating packets until the maximum size that can be sent in a radio frame is reached.

A UDS network has a star-shaped network topology, in which all clients are connected to a host. In normal unicast communication, transmissions from the host to a client or from a client to a host are performed in one pass, but transmissions from one client to another are performed in two passes because they are routed through the host. If you are sending the same data to multiple nodes, you can reduce the number of packets sent over the network as a whole by using broadcast communication. Broadcast communication sends data to all nodes connected to the network in a single pass. No matter how many nodes there are, it is possible to send them data with a single transmission. However, although it is guaranteed that all nodes will be in the communication range of a broadcast communication from the host, you must account for the possibility that some nodes will be out of range of a broadcast communication from a client.

#### Implementing a Mesh Network With UDS Communication

It is relatively easy to implement communication processing in a mesh network because all nodes in such a network perform the same processing. In a mesh network, however, the number of packets sent per unit of time over the entire network increases exponentially with the increase in communicating systems, because all nodes in a mesh network send packets to all other nodes.

With UDS communication, you can reduce the ratio of packets sent to nodes by using broadcast communication, but when broadcasting from a client you must account for the possibility that packets may not reach all nodes. Clients that detect they have not received a packet must then request that the packet be resent. Because the packet is resent using unicast communication, you must account for the latency of the round trip plus the traversal through the host.

With broadcast communication, the same packets ordinarily reach all nodes. Although this is fine if you want to send the same data to all nodes, this method of communication does not reduce the number of packets if you want to send different data to each node. If you specify `FORCE_DIRECT_BC` in the send options, unicast communication sends data in a single pass in the same manner as broadcast communication, and received nodes discard packets that are not addressed to them. You can reduce the number of packets sent over the entire network per unit of time by using this feature in conjunction with the feature to accumulate packets and send them in batches. If you use this method, however, you must account for the possibility that some nodes will be out of communication range.

## 3.1.5. Disconnecting

This section describes the procedure by which a host node disconnects client and spectator systems, and a client or spectator system disconnects itself.

## 3.1.5.1. Disconnecting Client and Spectator Systems as a Host Node

A host can disconnect client and spectator systems from a network using the following methods.

- Disconnect a specific client from the network.
- Disconnect all clients from the network.
- Disconnect all spectator systems from the network.

Call `nn::uds::EjectClient()` to disconnect a client from a network.

**Code 3-12. Disconnecting a Client From a Network**

```
nn::Result nn::uds::EjectClient(u16 nodeId);
```

To disconnect a specific client from a network, set *`nodeId`* equal to its node ID.

To disconnect all clients from a network at the same time, set *`nodeId`* equal to `BROADCAST_NODE_ID` (`0xFFFF`).

Call `nn::uds::EjectSpectator()` to disconnect spectator systems from a network.

**Code 3-13. Disconnecting Spectator Systems From a Network**

```
nn::Result nn::uds::EjectSpectator(void);
nn::Result nn::uds::AllowToSpectate(void);
```

Call `nn::uds::EjectSpectator()` to disconnect all spectator systems from a network and deny any further connections from spectator systems. This assumes that the network was initially created to allow spectator connections. Call `nn::uds::AllowToSpectate()` to again allow spectator connections.

## 3.1.5.2. Destroying a Network as a Host Node

A host can disconnect all client and spectator systems and then free a network. Call `nn::uds::DestroyNetwork()` to free the network.

**Code 3-14. Destroying a Network**

```
nn::Result nn::uds::DestroyNetwork(void);
```

**Note:** If the host destroys the network, the client receives a disconnect reason of `DISCARDED_FROM_NETWORK` (in the `disconnectReason` member of the

nn::uds::ConnectionStatus structure). In some cases, however, a reason of CONNECTION_LOST might be received depending on the communication environment.

## 3.1.5.3. Disconnecting as a Client or Spectator System

Call nn::uds::DisconnectNetwork() to disconnect a client or spectator system from its current network.

**Code 3-15. Disconnecting From a Network**

```
nn::Result nn::uds::DisconnectNetwork(void);
```

## 3.1.5.4. Controlling Connections From Clients by the Host

The host allows and disallows client connections to the network.

**Code 3-16. Controlling Connections From Clients**

```
nn::Result nn::uds::DisallowToConnect(bool isDisallowToReconnect = false);
nn::Result nn::uds::AllowToConnect();
```

The nn::uds::DisallowToConnect() function maintains current connections and prevents connections from further clients. However, if false is specified in *isDisallowToReconnect*, reconnection is allowed if the client is disconnected after this function is called. Client connections are allowed under initial settings when the network is configured.

The nn:uds::AllowToConnect() function allows connections from clients. Denial of spectator connections set using nn::uds::EjectSpectator() remains disallowed.

## 3.1.5.5. Disconnections Due to Transitioning to Sleep Mode or Switching to Wireless-Disabled Mode

If the system transitions to sleep mode without terminating UDS communication or the system is changed to wireless-disabled mode using wireless communication, the internal state of UDS communication changes to nn::uds::STATE_ERROR. If any function other than nn::uds::Finalize() is called in this state, it returns nn::uds:: ResultInvalidState if the system is in sleep mode, and nn::uds::ResultWirelessOff if the system is in wireless-disabled mode. If the internal state recovers from the error, UDS communication must be ended using nn::uds::Finalize(), and then restarted. If the system was disconnected by being

changed to wireless-disabled mode, communication must be reinitialized after it changes back to wireless-enabled mode.

> **Note:** When transitioning to sleep mode during UDS communication, we recommend that applications terminate UDS communication first and then enter sleep mode.

## 3.1.6. Finalizing

Call `nn::uds::Finalize()` to shut down UDS communication.

**Code 3-17. Shutting Down the UDS Library**

```
nn::Result nn::uds::Finalize(void);
```

The library frees the memory block region allocated by `nn::uds::Initialize()` in addition to the `nn::os::Event` instance. Applications that use UDS communication must finalize UDS communication before shutting down. For more information, see Cautions When Shutting Down in the *3DS Programming Manual: System*.

## 3.1.7. Network State Synchronization

The `uds` library automatically synchronizes the following information on nodes connected to a network.

- The current connection count
- Each node's node ID
- Each node's local Friend Code (scrambled for privacy protection)
- Each node's user name

The application does not need to explicitly send notifications when node information is updated.

This information is synchronized across all host and client nodes, so that clients can detect new connections and disconnections and also get information about each node.

## 3.1.8. Node ID Assignment

Nodes connected to a UDS communication network are managed by 16-bit numbers called *node IDs*. Host and client systems are assigned node IDs but spectator systems are not.

Unlike an ID in MP communications, a node ID cannot be assigned to another node after it has been used in a network. When a client reconnects to the same network, it is assigned the same node ID as when it last connected.

Because a node is assigned the same node ID when it reconnects, the host internally keeps information for 64 disconnected nodes. If more than this number of nodes are disconnected, information is deleted for the earliest client node to be disconnected. A node is assigned a different ID when it reconnects to a network from which its disconnected node information was deleted. The following node IDs are reserved and are never assigned to clients.

**Table 3-5. Reserved Node IDs**

| Reserved Node IDs | Description |
|---|---|
| 0 | A nonexistent node. Used for internal library processing. |
| 1 | This is a fixed value assigned to the host. |
| `0xFFFF (BROADCAST_NODE_ID)` | This node ID is specified when broadcasting. |

## 3.1.9. Getting Information

You can get a system's own information, such as connection state and signal strength.

### 3.1.9.1. Getting the Connection State

Call the `nn::uds::GetConnectionStatus()` function to get the system's current connection state.

**Code 3-18. Getting the Connection State**

```
nn::Result nn::uds::GetConnectionStatus(nn::uds::ConnectionStatus* pStatus);
```

Pass a pointer to an `nn::uds::ConnectionStatus` structure in the *`pStatus`* parameter to hold the obtained connection state. The following sample code defines the structure.

**Code 3-19. nn::uds::ConnectionStatus Structure Definition**

```
struct nn::uds::ConnectionStatus
{
    State               nowState;
    DisconnectReason    disconnectReason;
    u16                 myNodeId;
    bit16               updateNodeBitmap;
    u16                 nodeIdList[NODE_MAX];
    u8                  nowEntry;
    u8                  maxEntry;
```

```
    bit16                slotBitmap;
};
```

The `nowState` member stores the current state. For more information about states, see 3.1.1. Internal UDS Communication States.

The `disconnectReason` member stores the reason for disconnection. The following table shows the correlation between member values and reasons.

**Table 3-6. Disconnection Reasons**

| Value | Disconnection Reason |
|---|---|
| `BEFORE_COMMUNICATION` | No communication is established. |
| `NETWORK_IS_AVAILABLE` | Not disconnected. Connection is maintained. |
| `REQUEST_FROM_MYSELF` | Disconnected due to the local host's own operation. |
| `REQUEST_FROM_SYSTEM` | Disconnected due to a request from the Nintendo 3DS system, such as for transition to sleep mode or wireless-disabled mode. |
| `DISCARDED_FROM_NETWORK` | Disconnected due to an operation on the host system. |
| `CONNECTION_LOST` | Connection could not be maintained due to deteriorating signal conditions. |
| `UNKNOWN_DISCONNECT_REASON` | Disconnection reason unknown. |

The `myNodeId` member stores its own Node ID. In the case of `Spectator`, it stores the value `0xFFFF`.

The `updateNodeBitmap` member stores a bitmap indicating those nodes that have changed state since the previous function call. Bits of value 1 correspond to those node IDs for nodes that have had state changes. Bit 1 in the bitmap corresponds to the first element in `nodeIdList`, with the least-significant bits to most-significant bits corresponding to the first through last elements.

The `nowEntry` member stores the number of nodes currently connected to the network, and the `maxEntry` member stores the maximum number of nodes that can simultaneously connect. Note that `maxEntry` cannot be changed while communication is in progress.

The `slotBitmap` member stores a bitmap indicating the slots storing node information. This is not a delta, and reflects the current data state.

## 3.1.9.2. Getting the Signal Strength

Call the `nn::uds::GetLinkLevel()` function to get the system's current signal strength ("link level").

**Code 3-20. Getting the Signal Strength**

```
nn::Result nn::uds::GetLinkLevel(nn::uds::LinkLevel* pLinkLevel);
nn::uds::LinkLevel nn::uds::GetLinkLevel();
```

Pass a pointer to the variable for holding the signal strength value in the *pLinkLevel* parameter. The following table shows how link level values correspond to signal strength.

**Table 3-7. Signal Strength**

| Value | Description |
|---|---|
| LINK_LEVEL_0 | Signal strength is very low, or communication has not been established. |
| LINK_LEVEL_1 | Signal strength is low. |
| LINK_LEVEL_2 | Signal strength is medium. |
| LINK_LEVEL_3 | Signal strength is good. |

Applications do not need to constantly display the signal strength. Because this is a blocking function, we do not recommend calling it frequently (such as every game frame).

To display the signal strength as an indicator before connecting to a network as a client or spectator node, call `GetRadioStrength` on the instance of the `nn::uds::NetworkDescriptionReader` class obtained as the result of `nn::uds::Scan()`.

## 3.1.9.3. Getting the Node Information

Call the `nn::uds::GetNodeInformation()` function to get user information for the specified node.

**Code 3-21. Getting the Node Information**

```
nn::Result nn::uds::GetNodeInformation(nn::uds::NodeInformation* pNodeInfo,
                                       u16 nodeId);
```

*pNodeInfo* specifies a pointer to the `nn::uds::NodeInformation` structure storing the node information.

The *nodeId* argument specifies the ID of the node for which to get information.

## 3.1.9.4. Getting the Channel

Call the `nn::uds::GetChannel()` function to get the current channel.

**Code 3-22. Getting the Channel Information**

```
nn::Result nn::uds::GetChannel(u8* pChannel);
```

*pChannel* specifies a pointer to the variable that stores the channel value.

## 3.2. Nintendo 3DS Download Play

Nintendo 3DS Download Play (or simply "Download Play") is a feature used by hosts to distribute programs to clients through wireless communication, as in Nintendo DS Download Play. Nintendo 3DS Download Play has the following characteristics.

- Clients of up to 32 MB (33,554,432 Bytes) can be distributed.
- Only applications with client programs that run in standard mode can be distributed.
- If necessary, a system update is performed by wireless communication before distribution of client programs.
- A client program can get information that identifies the host during local communication.
- A client program is saved to a dedicated memory region, and is overwritten if another client program is downloaded.
- Client programs are not displayed on the HOME Menu (and cannot be started from the HOME Menu).

**Warning:** Nintendo 3DS Download Play cannot be performed between a development CTR system (debugger or development system) and a retail CTR system.

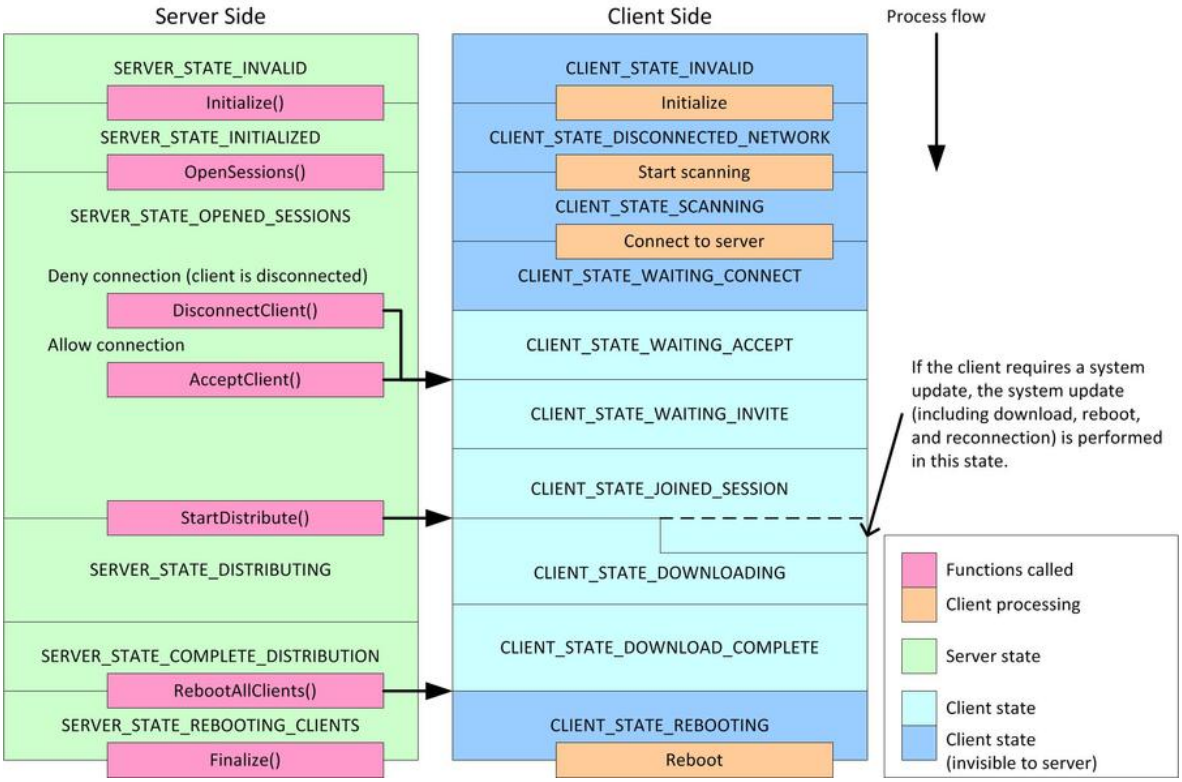
**Note:** The size of the client program is its size after being imported. To check this value, open the DevMenu's **SDMC** or **HIO** tab, and check the **RequiredSize** value when the client program is selected.

Use the DLP library provided by the CTR-SDK to support Download Play in your application.

The program for clients is provided by the system application, so for distribution, the application must prepare both the client program and the server code used to control how the program is distributed. This library also provides a class that supports features for clients acting as fake Download Play children. Because the library handles communication internally, the application simply needs to call the necessary functions to check the state.

The following figure shows the process flow on the server carrying out for Nintendo 3DS Download Play, based on the state transitions for the parent (server) and child (client).

**Figure 3-2. Process Flow Carried Out for Nintendo 3DS Download Play (Server)**




**Warning:** When Nintendo 3DS Download Play is performed with the downloadable application, the client on the lower system version of the development hardware may receive a fatal error after download is completed. The server version and the client's system version must be the same. For the production release of a card-based software application, the system update for the client takes place via wireless communications. For a downloadable application, communication is interrupted immediately after the system update is started.

## 3.2.1. Initializing

The `nn::dlp::Server` class has all the functions required for server code. You do not need to initialize the DLP library itself. Call the `nn::dlp::Server` class's `Initialize` function to initialize the server.

**Code 3-23. Initializing**

```
static nn::Result nn::dlp::Server::Initialize(
                bool* pNotice, nn::Handle eventHandle, u8 maxClientNum,
                u8 childIndex, void* pBuffer, size_t bufferSize,
                size_t blockBufferSize = MIN_NETWORK_BLOCK_BUFFER_SIZE * 2,
                size_t blockBufferNum = MIN_NETWORK_BLOCK_BUFFER_NUM);
static size_t nn::dlp::Server::GetBufferSize(u8 maxClientNum,
                size_t blockBufferSize = MIN_NETWORK_BLOCK_BUFFER_SIZE * 2,
                size_t blockBufferNum = MIN_NETWORK_BLOCK_BUFFER_NUM);
```

*pNotice* indicates whether the Download Play warning message must be displayed. When `true` is returned, display the message provided in the guidelines. The message indicates that when a downloadable application becomes a server, connections from clients with earlier system versions will be disconnected without a warning. Note that, for future updates, if the system supports system updates via a downloadable application, message display will no longer be required and `true` will never be returned in *pNotice*. To enable switching between displaying and not displaying a message according to the state of system updates, always display a message according to the result in *pNotice*. The *pNotice* parameter is never set to `true` for card-based applications. However, if a message is displayed according to *pNotice* in a card-based application, no source code modifications will be needed if it is later released as a downloadable application.

For *eventHandle*, pass the handle in the `nn::os::Event` class, which waits for notifications from Download Play. The application must initialize the events that pass this handle.

For *maxClientNum*, specify the maximum number of clients that can connect to the server. This value is between 1 and `MAX_CLIENT_NUM`.

For *childIndex*, specify the index of the client program to be distributed (`ChildIndex` in the RSF file). Icons and titles displayed on the client are obtained from header information in the client program specified by this parameter.

For the *pBuffer* and *bufferSize* parameters, specify the work buffer and its size. Pass the block buffer size and number of buffers in *blockBufferSize* and *blockBufferNum*. The block buffer is a client program read-out buffer. The application must allocate the work buffer from a location that is **not in device memory**, starting at an address that is `nn::dlp::Server::BUFFER_ALIGNMENT` (4096-byte) aligned. The buffer size must be a multiple of `nn::dlp::Server::BUFFER_UNITSIZE` (4096 bytes) that is greater than or equal to the size obtained from the `GetBufferSize()` function passing *maxClientNum*, *blockBufferSize*, and *blockBufferNum* as arguments. If an out-of-range value is specified for the argument, `GetBufferSize()` returns an indefinite value.

The `ServerWithName` class adds the ability to specify the user name to the `Initialize()` function of the `Server` class. If this class is used to specify a user name, always check for profanity according to the UGC guidelines. If the user name contains profanity according to the check, set `isNgUserName` to `true` in the structure passed to the `Initialize()` function, and pass the user name to the function as is, without processing to remove or overwrite the profanity. The `ServerWithName` class is the same as the `Server` class, except for the `Initialize()` function.

The following table shows the return values that can be returned by the `Initialize()` function and their causes.

**Table 3-8. Return Values Returned by the Initialize() Function**

| Return Value | Cause |
|---|---|
| `IsSuccess()` returns `true` | The process was successful. |
| `ResultAlreadyOccupiedWirelessDevice` | The wireless communication module is secured by another wireless process. Terminate infrastructure communication or other communications. |
| `ResultOutOfRange` | Out-of-range value specified in argument. |

| | |
|---|---|
| ResultInvalidPointer | Invalid pointer specified in argument. |
| ResultInvalidHandle | Invalid handle specified in argument. |
| ResultInternalError | Unrecoverable error in library. |
| ResultFailedToAccessMedia | Either the client program having the specified index does not exist, or the server card has been removed. |
| ResultChildTooLarge | The size of the client program exceeds the maximum size that can be distributed. |
| ResultInvalidRegion | The regions of the host and client programs differ. |
| ResultWirelessOff | Wireless communication unavailable (sleep mode or wireless-disabled mode). |

## 3.2.2. Checking the State

There are two ways to check the server state. You can either call the `GetState()` function to get it periodically, or you can call the `GetEventDesc()` function to get event details when the event specified during initialization enters the signaled state.

**Code 3-24. Checking the State**

```
static nn::Result nn::dlp::Server::GetEventDesc(
                                    nn::dlp::EventDesc* pEventDesc);
static nn::Result nn::dlp::Server::GetState(nn::dlp::ServerState* pState);
```

The `GetEventDesc()` function can get details for up to 32 events for storage in a queue, discarding the earliest items when the queue becomes full. Processing returns immediately with a return value of `nn::dlp::ResultNoData` in the absence of an event.

However, this does not mean that a server cannot function without monitoring events' signaled states. It is probably easier to implement server code that calls `GetState()` periodically to check the state and then process it appropriately.

**Note:** For this server implementation we strongly recommend polling for the state using `GetState()`. This makes error handing easier when errors arise, which invariably happens if it becomes impossible to sustain the communications state, for example, if the system enters sleep mode or switches to wireless-disabled mode while downloading or being invited to participate in a session.

The following table shows the server states that can be obtained by the `GetState()` function.

**Table 3-9. Server States**

| Definition | Description |
|---|---|
| SERVER_STATE_INVALID | Not initialized yet. |

| | |
|---|---|
| SERVER_STATE_INITIALIZED | Server initialization has completed. |
| SERVER_STATE_OPENED_SESSIONS | The server is accepting requests to join a session. Clients can join only in this state. |
| SERVER_STATE_DISTRIBUTING | The server is distributing. |
| SERVER_STATE_COMPLETE_DISTRIBUTION | The server has finished distributing. |
| SERVER_STATE_REBOOTING_CLIENTS | The server is rebooting the clients after distributing. Confirm that all clients have been disconnected before you finalize Nintendo 3DS Download Play. |
| SERVER_STATE_ERROR | An error has occurred, requiring reinitialization. The server transitions to this state if the system has entered sleep mode or switched to wireless-disabled mode some time between the `Initialize()` function and the `Finalize()` function, if the client has disconnected during distribution for any reason other than to reboot after distribution, or if the card has been removed in the middle of the process. |

The following table shows the return values that might be returned by the `GetEventDesc()` or `GetState()` function, and their causes.

**Table 3-10. Return Values Returned by the GetEventDesc() and GetState() Functions**

| Return Value | Cause |
|---|---|
| `IsSuccess()` returns `true` | The process was successful. |
| `ResultNoData` | No data to be obtained. |
| `ResultInvalidPointer` | Invalid pointer specified in argument. |

## 3.2.3. Starting and Stopping Sessions

In Nintendo 3DS Download Play, a *session* starts after distribution requests can be received from clients and ends when the distribution has completed and the clients restart.

You can use the `OpenSessions()` or `CloseSessions()` function to start or stop sessions.

**Code 3-25. Starting and Stopping Sessions**

```
static nn::Result nn::dlp::Server::OpenSessions(
                                  bool isManualAccept = false, u8 channel = 0);
static nn::Result nn::dlp::Server::CloseSessions();
```

Start a session with `true` for the *`isManualAccept`* parameter to have the application handle accepting or rejecting clients that attempt to join the session. Omit this argument or pass `false` to automatically accept connections.

Specify any one of 0, 1, 6, or 11 in *`channel`* for the communication channel to be used. If 0 is

specified, the channel is selected automatically. Usually, no channel needs to be specified.

> **Warning:** The channel is always selected automatically when executed on production version hardware.

If a client disconnects in the middle of a session or if, for another reason, the server state transitions stop, call the `CloseSessions()` function to stop the session. Because clients that have rebooted following distribution will not necessarily reconnect to the server, you must implement your server code to allow the user to stop a session at any time.

The following table shows the return values that might be returned by `OpenSessions()` or `CloseSessions()`, and their causes.

**Table 3-11. Return Values of the OpenSessions() and CloseSessions() Functions**

| Return Value | Cause |
|---|---|
| `IsSuccess()` returns `true` | The process was successful. |
| `ResultInvalidState` | The function called an invalid state. |
| `ResultOutOfRange` | Out-of-range value specified in argument. |
| `ResultWirelessOff` | Wireless communication unavailable (sleep mode or wireless-disabled mode). |

## 3.2.4. Checking the Connection State

After you have started a session and as long as no errors have occurred, use the `GetConnectingClients()` function to check both the number of connected clients and their node IDs. Use these IDs to get the client information and state with the `GetClientInfo()` and `GetClientState()` functions.

**Code 3-26. Checking the Connection State**

```
static nn::Result nn::dlp::Server::GetConnectingClients(
                        u16* pNum, u16 clients[], u16 size);
static nn::Result nn::dlp::Server::GetClientInfo(
                        nn::dlp::NodeInfo* pClientInfo, u16 nodeId);
static nn::Result nn::dlp::Server::GetClientState(
                        nn::dlp::ClientState* pClientState, u16 nodeId);
static nn::Result nn::dlp::Server::GetClientState(
                        nn::dlp::ClientState* pClientState, size_t* pTotalNum,
                        size_t* pDownloadedNum, u16 nodeId);
```

The `GetConnectingClients()` function stores the number of clients connected to a session in the *pNum* parameter and a list of their node IDs in *pClients*. Note that *pNum* is a pointer to a u16 variable and *clients* is a u16 array. Set *size* equal to the size of the array specified by

*pClients*. The array size must be greater than or equal to the value specified for *maxClientNum* during initialization.

Use the *nodeId* parameter to the GetClientInfo() or GetClientState() function to specify the node ID of the client for which to get information. If you specify a nonexistent client or server node ID, the function returns nn::dlp::ResultNoData.

The nn::dlp::NodeInfo structure specified by the *pClientInfo* parameter of the GetClientInfo() function stores the same node information mentioned in 3.1.9.3. Getting the Node Information. You can use this information to display user names and for other purposes.

The nn::dlp::ClientState-type pointer specified by the *pClientState* parameter to the GetClientState() function stores the client state. *pTotalNum* holds the total number of packets of the client program, and *pDownloadNum* holds the number of downloaded packets. Be careful with division for showing progress, because the value of *pTotalNum* can be *0*.

The client state is stored in *pClientState*.

**Table 3-12. Client State (Only When It Can Be Checked by the Server)**

| Definition | Description |
|---|---|
| CLIENT_STATE_WAITING_INVITE | Accepted by the server to participate in a session and waiting to be invited by the host to participate in a session. |
| CLIENT_STATE_WAITING_ACCEPT | The client is connected to the network and waiting for the server to accept participation in the session. |
| CLIENT_STATE_JOINED_SESSION | Joined in the session. |
| CLIENT_STATE_DOWNLOADING | Distribution has started and downloading is in progress. Note that a distribution may have just started and no data may have been received from the server yet. |
| CLIENT_STATE_DOWNLOAD_COMPLETE | Downloading has completed. |

The following table shows the return values that might be returned by the GetConnectingClients(), GetClientInfo(), or GetClientState() functions, and their causes.

**Table 3-13. Return Values Returned by the GetConnectingClients(), GetClientInfo(), and GetClientState() Functions**

| Return Value | Cause |
|---|---|
| IsSuccess() returns true | The process was successful. |
| ResultInvalidState | The function called an invalid state. |
| ResultInvalidPointer | Invalid pointer specified in argument. |
| ResultNoData | Not connected to a client with the specified node ID. |

## 3.2.5. Accepting and Rejecting Connections

After the server has started a session, it accepts client requests to join the session until it starts distribution. An application can accept or reject client connections if the session was started with `true` passed as the *`isManualAccept`* argument.

**Code 3-27. Accepting and Rejecting Connections**

```
static nn::Result nn::dlp::Server::AcceptClient(u16 nodeId);
static nn::Result nn::dlp::Server::DisconnectClient(u16 nodeId);
```

You can accept connections with `AcceptClient()` and reject connections with `DisconnectClient()`. However, these functions only work when the client state is `CLIENT_STATE_WAITING_ACCEPT`.

The following table shows the return values that might be returned by `AcceptClient()` or `DisconnectClient()`, and their causes.

**Table 3-14. Return Values Returned by the AcceptClient() and DisconnectClient() Functions**

| Return Value | Cause |
|---|---|
| `IsSuccess()` returns `true` | The process was successful. |
| `ResultInvalidState` | The function called an invalid state. |
| `ResultNoData` | Not connected to a client with the specified node ID. |
| `ResultWirelessOff` | Wireless communication unavailable (sleep mode or wireless-disabled mode). |

## 3.2.6. Starting Distribution

After you have established the clients to distribute to, you can call the `StartDistribute()` function to stop accepting requests to join the session and start distribution to clients.

**Code 3-28. Starting Distribution**

```
static nn::Result nn::dlp::Server::StartDistribute();
```

With this function, the system starts distributing and transitions to the `SERVER_STATE_PREPARING_FOR_TITLE_DISTRIBUTION` state.

Simply check the client connection states, display progress, and otherwise wait for the server state to transition to `SERVER_STATE_COMPLETE_DISTRIBUTION`, signaling that the distribution is complete.

**Warning:** Clients whose state does not change from `CLIENT_STATE_JOINED_SESSION` even after distribution has started are clients that require a system update. These clients

remain in this state while they receive the system update (a process which includes download, reboot, and reconnection), and until the client state transitions to `CLIENT_STATE_DOWNLOADING`, the server simply waits for the client to reconnect. Not all clients reconnect, however, because communication between the server and client may have become impossible (due to, for example, canceling a download, switching to wireless-disabled mode, degradation of communication quality, and so on). For this reason, make sure that you implement a solution so that user operations can stop a session at any time (using the `CloseSessions()` function).

The following table shows the return values that might be returned by the `StartDistribute()` function, and their causes.

**Table 3-15. Return Values Returned by the StartDistribute() Function**

| Return Value | Cause |
|---|---|
| `IsSuccess()` returns `true` | The process was successful. |
| `ResultInvalidState` | The function was called in an inappropriate state, or no client exists to be distributed. |
| `ResultWirelessOff` | Wireless communication unavailable (sleep mode or wireless-disabled mode). |
| `ResultFailedToAccessMedia` | Failed to access media. The server card may have been removed. |

**Note:** The specifications state that the client does not transition to the disconnected screen even if the connection is lost during distribution (for example, if the server card is pulled out). The application also does not need to deal with this situation.

## 3.2.7. Rebooting Clients

After distribution has finished and the server state has transitioned to `SERVER_STATE_COMPLETE_DISTRIBUTION`, call the `RebootAllClients()` function to reboot all clients that have joined the session.

**Code 3-29. Rebooting Clients**

```
static nn::Result nn::dlp::Server::RebootAllClients(
                        const char passpharase[] = NULL);
```

For *passphrase*, specify the passphrase used when the network was created, because it is needed to conduct local communications after clients reboot. You can specify a string as long as `MAX_CHILD_UDS_PASSPHRASE_LENGTH`, counting the `NULL` character. Specify a different value for this passphrase for each download session, and do not specify strings that are easy to guess.

When this function is called after distribution completes, all clients are restarted and the server state transitions to SERVER_STATE_REBOOTING_CLIENTS. Finalize the server after you have checked the clients' connection states and confirmed that they have all disconnected.

The following table shows the return values that might be returned by the RebootAllClients() function, and their causes .

**Table 3-16. Return Values Returned by RebootAllClients()**

| Return Value | Cause |
|---|---|
| IsSuccess() returns true | The process was successful. |
| ResultInvalidState | The function called an invalid state. |

## 3.2.8. Finalizing

After confirming that every client has rebooted and disconnected from the session, you can call the Finalize() function to finalize the server. The Finalize() function must be called when transiting to sleep status due to closing the system.

**Code 3-30. Finalization**

```
static nn::Result nn::dlp::Server::Finalize();
```

Nintendo 3DS Download Play ends when finalization is complete.

To perform local communication with a distributed client program, create a network using the UDS library after finalization and wait for clients to connect. However, note that clients do not necessarily reconnect.

The following table shows the return values that might be returned by Finalize(), and their causes.

**Table 3-17. Return Values Returned by Finalize()**

| Return Value | Cause |
|---|---|
| IsSuccess() returns true | The process was successful. |

## 3.2.9. Client Programs

After downloading a client program through Download Play, a client reboots itself as instructed by the server. To make it easier for the client program to perform local communication with the host that used to be the server, you can get information for reconnecting to this host from the nn::dlp::GetRebootInfo() function.

**Code 3-31. Getting Reconnection Information**

```
nn::Result nn::dlp::GetRebootInfo(nn::dlp::RebootInfo* pRebootInfo);
```

You can call this function without initializing the DLP library. Pass *pRebootInfo* a pointer to a `nn::dlp::RebootInfo` structure that stores reconnection information.

**Code 3-32. Definition of the nn::dlp::RebootInfo Structure**

```
typedef struct
{
    u8      bssid[6];
    char    passphrase[MAX_CHILD_UDS_PASSPHRASE_LENGTH];
    NN_PADDING1;
} nn::dlp::RebootInfo;
```

The BSSID and passphrase of the network to be created by the parent are stored in `bssid` and `passphrase`.

Based on this information, the application can automatically determine which network to connect to.

## 3.2.9.1. Embedding a Client Program

The unique IDs for the child program and the application on the server as specified in the RSF file must match. In addition, the client program must have a `Category` of `DlpChild`. Although you need a CTR icon for your client program, you do not need a CTR title banner for it.

If you are using `OMake` to build the application, specify the client program's CIA file in the `CHILD_APPS[]` build variable in the `OMakefile` for the application on the server.

If you are not using `OMake`, use the `ctr_makeciaarchive` command-line tool to create a CFA file, and include this in the CCI file by using the `–content` option of `ctr_makerom`.

## 3.2.9.2. Forcing a Download of the Client Program

Clients generally download only client programs that are either not present on the client or that are newer than previously imported versions, but you can use the Config tool to force a client to ignore the client program version and download it anyway.

## 3.2.9.3. Debugging a Client Program

Complete the following steps to debug a client program after Download Play has started the client program.

- Attach the client program by calling the debugger's ATTACHA command.
- Execution pauses, so use the LS command to load the debugging information from the AXF file.
- Resume execution.

### 3.2.9.4. Identifying Client Programs

Call the `nn::dlp::IsChild()` function to determine whether the currently running application is a client program.

**Code 3-33. Identifying Client Programs**

```
bool nn::dlp::IsChild();
```

Do not call this function when using the `nn::dlp::Server` or `nn::dlp::FakeClient` objects (the interval from first calling `Initialize()` until `Finalize()` completes).

## 3.2.10. Fake Clients

Fake clients (`nn::dlp::FakeClient` class) are provided so that applications can join Download Play sessions as if they were clients without actually downloading the client program or restarting.

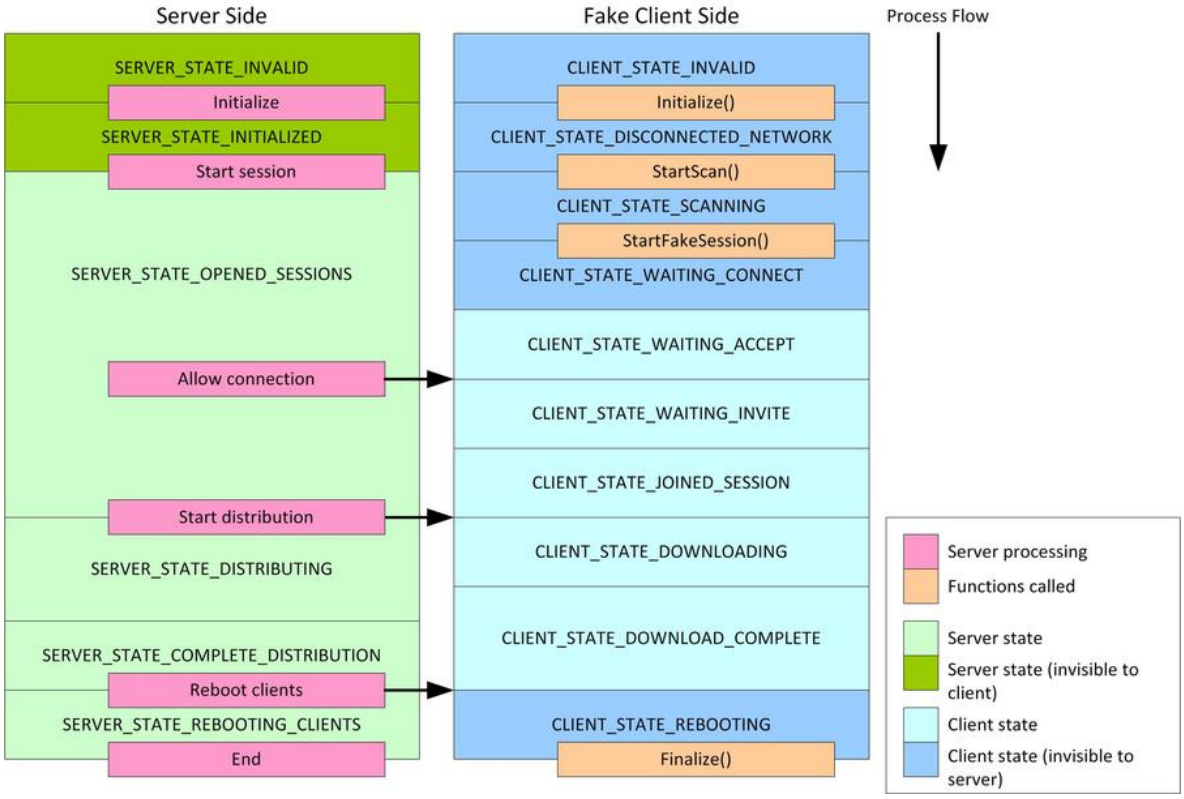
The use of fake clients enables applications to participate in the same session as the Nintendo 3DS systems that are participating in Download Play using smooth wireless communications in a network that includes both applications and client programs. The obvious advantage is that this allows the network to include users with and without the software. But it also enables session management and other tasks to be handled by the DLP library.

**Warning:** When revising a title that uses a fake client with an update or remaster version, special consideration is required if a fake client that differs from the remaster version will connect to the DLP server.

See the chapter about fake client features in the *Patch Manual*.

The following figure shows the process flow carried out on the fake client for Nintendo 3DS Download Play, based on the state transitions for the parent (server) and child (client).

**Figure 3-3. Process Flow of Download Play (Fake Client)**




## 3.2.10.1. Initializing

The `nn::dlp::FakeClient` class has all the functions required for fake client code. You do not need to initialize the DLP library itself. Just call the class's `Initialize()` function to initialize the fake client.

**Code 3-34. Initializing the Fake Client**

```
static nn::Result nn::dlp::FakeClient::Initialize(
u8 scanNum, nn::Handle eventHandle,
void* pBuffer, const size_t bufferSize);
static size_t nn::dlp::FakeClient::GetBufferSize(u8 scanNum);
```

For *`scanNum`*, specify the maximum number of titles that can be obtained in one scanning. Set this in the range from 1 to `MAX_SCAN_NUM`.

For *`eventHandle`*, pass the handle in the `nn::os::Event` class, which waits for notifications from Download Play. The application must initialize the events that pass this handle.

For the *`pBuffer`* and *`bufferSize`* parameters, pass the work buffer and its size. The application must allocate the work buffer from a location that is **not in device memory**, starting at an address that is `nn::dlp::FakeClient::BUFFER_ALIGNMENT` (4096-byte) aligned. The buffer size must be a multiple of `nn::dlp::FakeClient::BUFFER_UNITSIZE` (4096 bytes) that is greater than or equal to the size obtained from the `GetBufferSize()` function passing *`scanNum`* as an argument. If an out-of-range value is specified for the argument, `GetBufferSize()`

returns an indefinite value.

The FakeClientWithName class adds the ability to specify the user name to the Initialize() function of the FakeClient class. If this class is used to specify a user name, always check for profanity according to the UGC guidelines. If the user name contains profanity according to the check, set isNgUserName to true in the structure passed to the Initialize() function, and pass the user name to the function as is, without processing to remove or overwrite the profanity. The FakeClientWithName class is the same as the FakeClient class, except for the Initialize() function.

The following table shows the return values that can be returned by the Initialize() function, and their causes.

**Table 3-18. Return Values Returned by the Initialize() Function**

| Return Value | Cause |
|---|---|
| IsSuccess() returns true | The process was successful. |
| ResultAlreadyOccupiedWirelessDevice | The wireless communication module is secured by another wireless process. Terminate infrastructure communication or other communications. |
| ResultOutOfRange | Out-of-range value specified in argument. |
| ResultInvalidPointer | Invalid pointer specified in argument. |
| ResultInvalidHandle | Invalid handle specified in argument. |
| ResultInternalError | Unrecoverable error in library. |
| ResultWirelessOff | Wireless communication unavailable (sleep mode or wireless-disabled mode). |

## 3.2.10.2. Checking the State

There are two ways to check the state of the fake client. You can use the GetEventDesc() function to get event details when the event specified during initialization enters the signaled state, or you can use the GetMyStatus() function to get the state periodically.

**Code 3-35. Checking the Fake Client's State**

```
static nn::Result nn::dlp::FakeClient::GetEventDesc(
                        nn::dlp::EventDesc* pEventDesc);
static nn::Result nn::dlp::FakeClient::GetMyStatus(
                        nn::dlp::ClientStatus* pStatus);
```

The GetEventDesc() function can get details for up to 32 events for storage in a queue, discarding the earliest items when the queue becomes full. If there are no events, processing returns immediately with a return value of nn::dlp::ResultNoData.

Monitoring for the signaled states of events is not necessary for fake clients to work. It is probably easier to implement code that calls GetMyStatus() periodically (for example, every

game frame) to check the state and then process it appropriately.

> **Note:** For this fake client implementation we strongly recommend polling for the state using `GetMyStatus()`. This makes error handing easier when errors arise. This invariably happens if it becomes impossible to sustain the communications state; for example, if the system enters sleep mode or switches to wireless-disabled mode while scanning for servers or downloading.

The `nn::dlp::ClientStatus` structure obtainable with `GetMyStatus()` is defined as follows.

**Code 3-36. Definition of nn::dlp::ClientStatus Structure**

```
typedef struct
{
    u16             nodeId;
    SessionType     sessionType;
    ClientState     state;
    size_t          totalNum;
    size_t          downloadedNum;
} nn::dlp::ClientStatus;
```

The state of the fake client is stored in the `state` member. Table 3-19 provides the definitions for the stored values.

**Table 3-19. Fake Client States (Only Those States the Fake Client Can Check)**

| Definition | Description |
|---|---|
| `CLIENT_STATE_INVALID` | Not initialized yet. |
| `CLIENT_STATE_DISCONNECTED_NETWORK` | Initialization completed, or disconnected from the server. |
| `CLIENT_STATE_SCANNING` | Scanning for servers. |
| `CLIENT_STATE_WAITING_CONNECT` | Waiting to connect to the network. |
| `CLIENT_STATE_WAITING_INVITE` | Waiting to be invited to join the session. |
| `CLIENT_STATE_JOINED_SESSION` | Joined in the session. |
| `CLIENT_STATE_DOWNLOADING` | Distribution has started and downloading is in progress. Fake clients do not receive data from the server. |
| `CLIENT_STATE_DOWNLOAD_COMPLETE` | Downloading has completed. |
| `CLIENT_STATE_RECONNECTING_NETWORK` | Fake clients do not transition to this state. |
| `CLIENT_STATE_REBOOTING` | Reboot request received from the server. |
| `CLIENT_STATE_ERROR` | An error has occurred that requires reinitialization. The server transitions to this state if the system entered sleep mode or switched to wireless-disabled mode sometime between the `Initialize()` function and the `Finalize()` function, it is disconnected from the server during distribution, or if another client was disconnected from the server during distribution. |

The values stored in members other than `state` do not need to be implemented for fake clients.

## 3.2.10.3. Scanning for Servers

To get a list of servers (titles), the fake client scans for servers that have started Download Play sessions.

**Code 3-37. Scanning for Servers**

```
static nn::Result nn::dlp::FakeClient::StartScan(
                u32 uniqueId, u8 childIndex, const u8* pMac = NULL);
static nn::Result nn::dlp::FakeClient::StopScan();
```

Call the `StartScan()` function to start scanning for servers. You can narrow down the scan target titles and servers by specifying values for *`uniqueId`*, *`childIndex`*, and *`pMac`*.

> **Warning:** If you select the server for another application, the operations of that server may be adversely affected. For fake clients you must specify values for *`uniqueId`* and *`childIndex`* and scan only for those servers distributing specific titles.

Scanning for servers continues until you call the `StopScan()` function. The scan result for as many servers as specified by the *`scanNum`* parameter of the `Initialize()` function is retained until the next call to `StartScan()`.

The following table shows the return values that can be returned by the `StartScan()` function, and their causes.

**Table 3-20. Return Values Returned by the StartScan() Function**

| Return Value | Cause |
|---|---|
| `IsSuccess()` returns `true` | The process was successful. |
| `ResultInvalidState` | Function called in invalid state (in some state other than `CLIENT_STATE_DISCONNECTED_NETWORK`). |

## 3.2.10.4. Getting the Scan Results

To get the results of the scanning, call the `GetTitleInfo()` function and get the results in the form of a list of title information (`nn::dlp::TitleInfo`). Based on this title information, you can use the `GetServerInfo()` function to get the server information (`nn::dlp::ServerInfo`). These functions return `nn::dlp::ResultInvalidState` if they are called when the state of the

fake client is either CLIENT_STATE_INVALID or CLIENT_STATE_ERROR.

**Code 3-38. Getting the Scan Results**

```
static nn::Result nn::dlp::FakeClient::GetTitleInfo(
        nn::dlp::TitleInfo* pTitleInfo, bool isReset = false);
static nn::Result nn::dlp::FakeClient::GetTitleInfo(
        nn::dlp::TitleInfo* pTitleInfo,
        const u8* pMac, u32 uniqueId, u8 childIndex);
static nn::Result nn::dlp::FakeClient::GetServerInfo(
        nn::dlp::ServerInfo* pServerInfo, const u8* pMac);
static nn::Result nn::dlp::FakeClient::DeleteScanInfo(const u8* pMac);
```

Among the overloaded GetTitleInfo() functions, those that have the *isReset* parameter are used to get the list of title information. Normally you set *isReset* to false to get title information that has not yet been obtained by the GetTitleInfo() function. When IsSuccess() on the instance of the return value returns true, new title information has been obtained. If there is no not-yet-obtained title information or if no more title information can be saved, nn::dlp::ResultNoData is returned. By specifying true for *isReset* you can get the whole list of title information again from the start, including title information that has already been obtained. Scanning for servers continues from its start with StartScan() to its end with StopScan(), and you can store title information on as many titles as specified by the *scanNum* parameter of the Initialize() function. The list of title information is cleared the next time StartScan() is called.

The overloaded function with the *pMac*, *uniqueId*, and *childIndex* parameters is used when you know the server's MAC address, unique ID, and client program index.

The nn::dlp::TitleInfo structure is defined as follows.

**Code 3-39. Definition of nn::dlp::TitleInfo**

```
typedef struct
{
    u32                         uniqueId;
    u8                          childIndex;
    NN_PADDING3;
    u8                          mac[6];
    u16                         programVersion;
    bit8                        ratingInfo[RATING_INFO_SIZE];
    wchar_t                     shortTitleName[SHORT_TITLE_NAME_LENGTH];
    wchar_t                     longTitleName[LONG_TITLE_NAME_LENGTH];
    nn::dlp::IconInfo           icon;
    u32                         importSize;
    nn::cfg::CfgRegionCode      region;
    bool                        ulcd;
    NN_PADDING2;
} nn::dlp::TitleInfo;
```

Use the three members uniqueId, childIndex, and mac to specifically identify title information.

Use the GetServerInfo() function to get server information. For *pMac*, specify the MAC address of the server from which you want to get server information. If there is no device with that specified MAC address or if the device is not operating as a Download Play server, the

function returns `nn::dlp::ResultNoData`.

The server information includes information about link levels and a list of connected clients. The structure is defined as follows.

**Code 3-40. Definition of nn::dlp::ServerInfo**

```
typedef struct
{
    u8                      mac[6];
    u8                      channel;
    nn::uds::LinkLevel      linkLevel;
    u8                      maxNodeNum;
    u8                      nodeNum;
    bit16                   dlpVersion;
    NN_PADDING4;
    NodeInfo                nodeInfo[MAX_NODE_NUM];
    nn::os::Tick            lastUpdateTick;
} nn::dlp::ServerInfo;
```

If the state is `CLIENT_STATE_SCANNING` (meaning that scanning is currently taking place), the values for `linkLevel` (the link level), `nodeNum` (the number of connected nodes, including the server), and `nodeInfo` (details about the nodes) all reflect the current state. In the other states there are no changes, so in these cases use the functions that are introduced in 3.2.10.6. Getting Extraneous Information.

Download Play servers are invariably assigned 1 as their node ID, just like the hosts in Local Play communications. So that several servers can be listed even when scanning has been narrowed down, consider measures such as showing user names (`userName.userName`) for nodes that have 1 as their node ID (`nodeId`) in the node information (`nodeInfo`) of the server information. If `isNgUserName` in the user name information (`username`) is `true`, the user name contains profanity. In that case, you must follow the Guidelines about profanity.

To delete specific sets of title information from the scan results, call the `DeleteScanInfo()` function. The scan results are deleted for the server with the MAC address specified by *`pMac`*. An example of when to use this is when information about a server has not been updated for a specified period of time. If there is no server with the specified MAC address, the function returns `nn::dlp::ResultNoData`. The function returns `nn::dlp::ResultInvalidState` if it is called when the state is something other than `CLIENT_STATE_DISCONNECTED_NETWORK` or `CLIENT_STATE_SCANNING`.

## 3.2.10.5. Joining a Session

To join a session, select a server for connection from the list of servers (title information) and call the `StartFakeSession()` function. From the server's point of view, the fake client looks just like the actual clients that are participating in the system application's Download Play session, but the fake client does not download the client program.

> **Note:** Before joining the session you must call `StopScan()` and stop scanning for servers.

**Code 3-41. Joining in Session**

```
static nn::Result nn::dlp::FakeClient::StartFakeSession(
                        const u8* pMac, u32 uniqueId, u8 childIndex);
static nn::Result nn::dlp::FakeClient::StopFakeSession();
```

The values specified for the *pMac*, *uniqueId*, and *childIndex* parameters can be obtained from the title information obtained by `GetTitleInfo()`.

The following table shows the return values that can be returned by the `StartFakeSession()` function, and their causes.

**Table 3-21. Return Values Returned by the StartFakeSession() Function**

| Return Value | Cause |
|---|---|
| `IsSuccess()` returns `true` | The process was successful. |
| `ResultInvalidState` | Function called in invalid state (in some state other than `CLIENT_STATE_DISCONNECTED_NETWORK`). |
| `ResultNoData` | The title as specified by *pMac*, *uniqueId*, and *childIndex* does not exist. |
| `ResultWirelessOff` | Wireless communication unavailable (sleep mode or wireless-disabled mode). |
| `ResultNotFoundServer` | The server as specified by *pMac* does not exist. |
| `ResultServerIsFull` | Cannot connect to the server because the maximum number of clients is already connected. Cannot connect unless the number of connected clients decreases. |
| `ResultDeniedFromServer` | Connection denied because the client program is being distributed by the server. |
| `ResultConnectionTimeout` | Connection was not established within a certain amount of time. This result can be returned when wireless reception is poor or when the communications load is heavy. |
| `ResultInternalError` | Error due to an inconsistency in the library. |

If the state transitions have stopped during a session, call the `StopFakeSession()` function to stop the session. The `IsSuccess()` method on the instance returned by this function invariably returns `true`. Just as is the case with servers, you must implement fake clients so that users can end sessions at any time.

## 3.2.10.6. Getting Extraneous Information

Several functions have been prepared for getting extraneous information that is not essential to the implementation of fake clients.

**Code 3-42. Getting Extraneous Information**

```
static nn::Result nn::dlp::FakeClient::GetConnectingNodes(
                        u8* pNum, u16* pNodeIds, u16 size);
static nn::Result nn::dlp::FakeClient::GetNodeInfo(
                        nn::dlp::NodeInfo* pNodeInfo, u16 nodeId);
static nn::Result nn::dlp::FakeClient::GetLinkLevel(
                        nn::uds::LinkLevel* pLinkLevel);
```

Use the `GetConnectingNodes()` function to get a list of nodes that are connected to a session.

Use the `GetNodeInfo()` function to get detailed information about the nodes that are connected to a session. The `nn::dlp::NodeInfo` returned to *`pNodeInfo`* is the same as the `nn::uds::NodeInfomation` in local communication. For more about this detailed node information, see 3.1.9.3. Getting the Node Information.

Use the `GetLinkLevel()` function to get the link level (the quality of the connection) with the server. For more information about the values returned to *`pLinkLevel`*, see 3.1.9.2. Getting the Signal Strength.

## 3.2.10.7. Ending Download

When the fake client has transitioned to the `CLIENT_STATE_DOWNLOAD_COMPLETE` state, it indicates only that the fake client itself has completed downloading. Other clients may still be downloading. The overall Download Play session has not completed yet, so wait until all clients connected to the server have finished downloading.

When all clients connected to the server have finished downloading, the server issues a reboot request to all clients. When the fake client receives this request, it transitions to the `CLIENT_STATE_REBOOTING` state. Only at this stage can you use the `GetPassphrase()` function to get the passphrase for local communications that was specified by the server.

After the passphrase has been obtained, call the `Finalize()` function and conduct the fake client finalization process.

**Code 3-43. Getting Passphrase and Finalizing**

```
static nn::Result nn::dlp::FakeClient::GetPassphrase(char* pPassphrase);
static nn::Result nn::dlp::FakeClient::Finalize();
```

For the *`pPassphrase`* parameter of `GetPassphrase()`, specify a buffer that is equal to or larger than `MAX_CHILD_UDS_PASSPHRASE_LENGTH` bytes. If the state is anything other than `CLIENT_STATE_REBOOTING`, the function returns `ResultInvalidState`.

`Finalize()` always succeeds. Consequently, the `IsSuccess()` method on the returned instance always returns `true`.

To conduct local communications with the parent (the server) at this stage, use the MAC address of the server that was selected when the session started and the passphrase that was obtained by the `GetPassphrase()` function.

Make sure that you call the `Finalize()` function and perform the finalization process when the state transitions to sleep mode and when the HOME Menu needs to start.

## 3.3. Automatic Connection

The AC (automatic connection) library is for automatically connecting to the Internet via the means specified in the system's network settings, specifically via a wireless LAN access point, the Nintendo Wi-Fi USB Connector, the Nintendo Zone, a retail access point, or a publicly available wireless LAN access point. There are no libraries for directly accessing the Nintendo 3DS system's wireless communication module, so you must use the `ac` library to establish an Internet (or LAN) connection to access external servers from a Nintendo 3DS unit.

The AC library accesses a wireless LAN access point or other network device according to the system's network settings. Consequently, valid wireless LAN access point connection settings must be configured ahead of time.

> **Note:** Instead of a menu for configuring the system settings, Nintendo provides the NetworkSetting tool to change network settings. The AC library also provides the `nn::ac::DebugSetNetworkSetting1()` debugging function for changing network setting 1, the `nn::ac::DebugSetNetworkArea()` function for specifying whether the automatically established connection need be all the way to the Internet or just to the access point, and the `nn::ac::DebugSetApType()` function for specifying the types of access points to connect to as a bitwise OR of `ApType` enumerators (see Table 3-22. nn::ac::ApType Enumerated Type).
>
> Note that these functions only work when the system is set to debugging mode (To set a system to debugging mode, open the Config tool, select **Debug Settings**, and set **Debug Mode** to **enable**).

If the AC library is used to establish an Internet connection, the daemon manager secures the wireless communication module in infrastructure communication mode. This may interfere with background communication such as is used for StreetPass connections.

AC library functions return their results as instances of the `nn::Result class`. `IsSuccess()` returns `true` if execution has completed successfully.

### 3.3.1. Initializing

Call the `nn::ac::Initialize()` function to initialize the AC library.

**Code 3-44. Initializing the AC Library**

```
nn::Result nn::ac::Initialize();
bool nn::ac::IsInitialized();
```

If nn::ac::ResultAlreadyInitialized is returned, the application has already initialized the AC library. Although this result is not an error, the Finalize() function must be called the same number of times as the Initialize() function is called because the library maintains a counter of the number of times initialization has been performed. Call nn::ac::IsInitialized() to determine whether the library has already been initialized.

## 3.3.2. Connecting Automatically

Automatic connections are established according to the conditions created by a call to the nn::ac::CreateDefaultConfig() function.

**Code 3-45. Creating Connection Conditions**

```
nn::Result nn::ac::CreateDefaultConfig(nn::ac::Config* config);
```

The connection conditions are stored in the nn::ac::Config structure passed in the *config* parameter. The application must allocate this structure. If NULL is specified in this parameter, nn::ac::ResultInvalidData is returned.

Start connecting automatically by passing the created connection conditions in the *config* parameter to the nn::ac::Connect() or nn::ac::ConnectAsync() functions.

**Code 3-46. Connecting Automatically**

```
nn::Result nn::ac::Connect(nn::ac::Config& config);
nn::Result nn::ac::ConnectAsync(nn::ac::Config& config, nn::os::Event* event);
nn::Result nn::ac::CancelConnectAsync();
nn::Result nn::ac::GetConnectResult();
bool nn::ac::IsConnected();
```

**Warning:** The EULA agreement is checked during automatic connection processing. The FS library must be initialized ahead of time.

The nn::ac::ConnectAsync() function is the asynchronous version of nn::ac::Connect(). When using the asynchronous version, the event passed to the *event* parameter enters the signaled state when the automatic connection attempt completes, regardless of whether the attempt was successful. After the event is signaled, make sure that you get the connection result by calling the nn::ac::GetConnectResult() function. Local communication fails to start and StreetPass does not operate until you get this result. Call the nn::ac::CancelConnectAsync() function to cancel a connection attempt that is in progress. The event passed to the

`nn::ac::ConnectAsync()` function's *event* parameter enters the signaled state when the connection is canceled or when the attempt succeeds before being canceled. `nn::ac::ResultCanceled` is returned as the execution result if execution is canceled.

> **Warning:** If the `nn::ac::CancelConnectAsync()` function is called when a background-communication connection request is being processed, it cancels the application's connection request and returns `nn::ac::ResultSuccess`.

> **Note:** The user-defined automatic connection target settings take priority over Nintendo Zone and hot spot settings. Because a connection priority order is not defined in the user settings, specifically designate one connection target in the user settings. The connection target can be specified with `nn::ac::DebugSetApType()` in debugging mode.

After the automatic connection attempt has succeeded, `nn::ac::ResultWanConnected` is returned as the execution result if an Internet connection has been established and `nn::ac::ResultLanConnected` is returned if a wireless LAN connection has been established. At this point, communication via the network is possible using socket communication or another wireless communication module. `nn::ac::ResultNotFoundAccessPoint` is returned if a connectable access point cannot be found.

> **Note:** Success may be returned immediately if a background-communication connection has already been established. Conversely, if a connection has not been established, the process could take some time, and it takes longer than normal when connecting to an access point that has not established an Internet connection.

If `nn::ac::ResultWifiOff` is returned as the execution result, the wireless communication module cannot be used because it is in wireless-disabled mode. If `nn::ac::ResultNotAgreeEula` is returned as the result, it could be because the EULA agreement check has failed or because an icon file has not been set in the application. For other execution results, see the *CTR-SDK API Reference*.

You can call the `nn::ac::IsConnected()` function to determine whether the application has connected to an access point through the automatic connection process. But even if it is determined that connection to an access point or another connection has been established through background communications, you still must have your application conduct the automatic connection process. If the application does not do this, the connection may be unexpectedly cut when the process connected in the background ends.

## 3.3.3. Getting the Connection State

Use the following functions to get the type of access point the system has automatically connected to, or the signal strength of the access point currently connected.

**Code 3-47. Getting the Connection State**

```
nn::Result nn::ac::GetConnectingApType(nn::ac::ApType* apType);
nn::Result nn::ac::GetConnectingNintendoZoneBeaconSubset(
                                nn::ac::NintendoZoneBeaconSubset* beacon);
nn::Result nn::ac::GetConnectingHotspotSubset(nn::ac::HotspotSubset* hotspot);
nn::Result nn::ac::GetLinkLevel(nn::ac::LinkLevel* linkLevel);
nn::ac::LinkLevel nn::ac::GetLinkLevel();
```

The `nn::ac::GetConnectingApType()` function returns the type of connected access point in the *apType* parameter. If called while unconnected, `nn::ac::ResultNotConnecting` is returned in the return value. Access point types are defined as `nn::ac::ApType` enumerators as shown in the following table. Access point types may be added in firmware updates, so note when implementing applications that you may get a state other than those listed.

**Table 3-22. nn::ac::ApType Enumerated Type**

| Enumerator | Access Point |
|---|---|
| AP_TYPE_NONE | None |
| AP_TYPE_USER_SETTING_1 | Network setting 1. |
| AP_TYPE_USER_SETTING_2 | Network setting 2. |
| AP_TYPE_USER_SETTING_3 | Network setting 3. |
| AP_TYPE_NINTENDO_WIFI_USB_CONNECTOR | Nintendo Wi-Fi USB Connector. |
| AP_TYPE_NINTENDO_ZONE | Nintendo Zone. |
| AP_TYPE_WIFI_STATION | Wi-Fi Station |
| AP_TYPE_FREESPOT | Free hot spot (consolidated into AP_TYPE_HOTSPOT). |
| AP_TYPE_HOTSPOT | Hot spot. |
| AP_TYPE_ALL | All of the above (`nn::ac::DebugSetApType()` only). |

The `nn::ac::GetConnectingNintendoZoneBeaconSubset()` function stores the Nintendo Zone beacon during connection in *beacon*. The `nn::ac::GetConnectingHotspotSubset()` function stores the hot spot information during connection in *hotspot*. Both functions return `nn::ac::ResultInvalidLocation` if called while connected to other than a supported access point.

Call `nn::ac::GetLinkLevel()` to get the signal strength of the currently connected access point. The return value or the signal strength returned in the *pLinkLevel* parameter is an `nn::ac::LinkLevel` enumerated type, as defined in the following table. If an error occurs while getting the wireless signal strength, this function returns LINK_LEVEL_0. The value of *pLinkLevel* does not need to be reassigned when an error occurs.

**Table 3-23. nn::ac::LinkLevel Enumerated Type**

| | |
|---|---|

| Enumerator | Signal Strength |
|---|---|
| LINK_LEVEL_0 | Signal is extremely weak, or communication has not been established. |
| LINK_LEVEL_1 | Weak signal. |
| LINK_LEVEL_2 | Medium-strength signal. |
| LINK_LEVEL_3 | Strong signal. |

There will times when AP_TYPE_WIFI_STATION or AP_TYPE_NINTENDO_ZONE is returned when connecting to a Nintendo 3DS station. This is a result of the distance between the devices.

Either nn::ac::AP_TYPE_WIFI_STATION, nn::ac::AP_TYPE_NINTENDO_ZONE, or nn::ac::AP_TYPE_HOTSPOT will be returned when connecting to a SpotPass hub. Use the bitwise OR from these results to determine whether a connection is being made to a SpotPass hub.

## 3.3.4. Disconnecting

After an automatic connection has been established, the system may disconnect either by the application explicitly disconnecting or by deteriorating signal conditions or other external factors.

**Code 3-48. Disconnecting**

```
nn::Result nn::ac::Close();
nn::Result nn::ac::CloseAsync(nn::os::Event* event);
nn::Result nn::ac::GetCloseResult();
nn::Result nn::ac::RegisterDisconnectEvent(nn::os::Event* event);
```

Call the nn::ac::Close() or nn::ac::CloseAsync() function to explicitly close an established connection of an application. The nn::ac::CloseAsync() function is the asynchronous version of nn::ac::Close(). When using the asynchronous version, the event passed to the *event* parameter enters the signaled state when the disconnection attempt completes, regardless of whether the attempt was successful. After the event is signaled, get the connection result by calling the nn::ac::GetCloseResult() function.

Whether the connection has been disconnected can only be determined by whether the IsSuccess() method on the instance of the returned execution result returns true. If connection is established automatically by a daemon running in the background, the status after connection is lost merely transitions to disconnected (nn::ac::STATUS_IDLE).

An established connection may also be disconnected due to deteriorating signal conditions, leaving the access point's effective range, or by another external factor. The application is notified of a disconnection by the signaling of the event passed in the *event* parameter to the nn::ac::RegisterDisconnectEvent() function. After establishing an automatic connection, create a thread that waits to be notified by the event passed to this function to detect the cause of a disconnection. The application does not need to take any special steps before exiting the library because the connection to the access point is guaranteed to be ended at the time the event is signaled.

### 3.3.5. Finalizing

When you are no longer using the AC library (such as when you are no longer using a network connection), call the `nn::ac::Finalize()` function to finalize the library.

**Code 3-49. Finalizing the AC Library**

```
nn::Result nn::ac::Finalize();
```

If this function is called in an uninitialized state, `nn::ac::ResultNotInitialized` (not an error) is returned.

## 3.4. Reliable Local (RDT) Communication

The CTR-SDK SDK provides the RDT library for Reliable local (RDT) communications.

The `rdt` library is a higher-level library that uses the `uds` library to provide the following features.

- Reliable, lossless data transmission over one-to-one connections.
- Transmitted byte streams are guaranteed to be received in the order sent.
- Unidirectional transmission from the sender to the receiver (bidirectional communication is possible by opening two one-way channels, but this doubles the memory and other resource requirements).

The RDT library is not designed for the regular sending of limited amounts of data, but rather can be used for sending larger amounts of data.

The RDT library uses the following terminology.

**Table 3-24. Terminology Used With the RDT Library**

| Term | Description |
|---|---|
| Sender | Class of the system sending data via RDT communication. |
| Receiver | Class of the system receiving data via RDT communication. |

### 3.4.1. Initializing

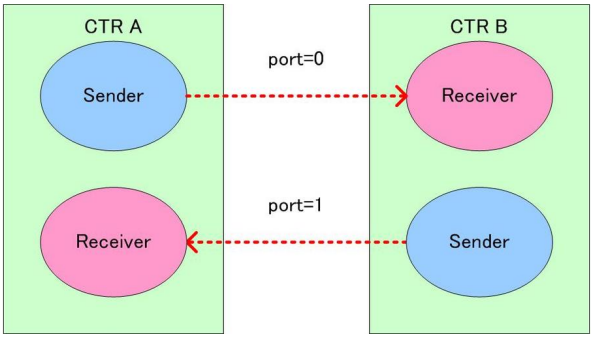
The RDT library uses UDS to implement reliable communication. You must establish a UDS network connection before you can use the RDT library. In other words, the host must have called

`nn::uds::CreateNetwork()`, and the client must have called `nn::uds::ConnectNetwork()`.

After establishing a network connection, initialize the Sender and Receiver. Create an instance of either the `nn::rdt::Sender` or `nn::rdt::Receiver` class, and call the `Initialize()` member function. A system may be either the Sender or Receiver depending on the direction of data transfer (regardless of host or client designations), so choose which class to instantiate accordingly.

When establishing two channels for bidirectional communication, specify different ports for a system's `Sender` and `Receiver` instances.

**Figure 3-4. Bidirectional Communication**




## 3.4.1.1. Initializing the Sender

Call the `Sender` object's `Initialize()` member function to initialize it. After it is initialized successfully, the `Sender` object implicitly creates two `nn::uds::EndpointDescriptor` instances. The function calls `nn::uds::Attach()` internally, implicitly allocating a reception buffer. As a result, initialization fails and the function returns `nn::uds::ResultOutOfMemory` if the specified reception buffer is too small when the UDS library is initialized.

**Code 3-50. Initializing the Sender**

```
nn::Result nn::rdt::Sender::Initialize(const nn::rdt::SenderConfig &config);
```

For the *`config`* parameter, specify an `nn::rdt::SenderConfig` structure storing all of the initialization parameters. The following sample code defines the structure.

**Code 3-51. SenderConfig Structure**

```
struct nn::rdt::SenderConfig
{
    void *pWorkBuf;
    void *pSendBuf;
    u16   sendBufSize;
    u16   nodeId;
    u8    port;
    u8    padding[3];
};
```

For the *pWorkBuf* member, specify the starting address of the working memory allocated for the Sender object. This working memory must be `nn::rdt::Sender::SENDER_WORKBUF_ALIGNMENT` (8-byte) aligned, and it must be at least as big as `nn::rdt::Sender::SENDER_WORKBUF_SIZE` (32,768 bytes).

For the *pSendBuf* and *sendBufSize* members, specify the starting address and size of the send buffer.

For the *nodeId* and *port* members, specify the node ID and port number for the Receiver system. These must be the same values as those specified for the UDS library.

Do not free the working memory and send buffer passed to `Initialize` until you call `Finalize()`.

## 3.4.1.2. Initializing the Receiver

Call the `Receiver` object's `Initialize()` member function to initialize it. After success, the `Receiver` object implicitly creates two `nn::uds::EndpointDescriptor` instances internally. The function calls `nn::uds::Attach()` internally, implicitly allocating a reception buffer. As a result, initialization fails and the function returns `nn::uds::ResultOutOfMemory` if the specified reception buffer is too small when the UDS library is initialized.

**Code 3-52. Initializing the Receiver**

```
nn::Result nn::rdt::Receiver::Initialize(
                        const nn::rdt::ReceiverConfig &config);
```

For the *config* parameter, specify an `nn::rdt::ReceiverConfig` structure storing all of the initialization parameters. The following sample code defines the structure.

**Code 3-53. ReceiverConfig Structure**

```
struct nn::rdt::ReceiverConfig
{
    void *pWorkBuf;
    void *pRecvBuf;
    u16  recvBufSize;
    u16  nodeId;
    u8   port;
    u8   padding[3];
};
```

For the *pWorkBuf* member, specify the starting address of the working memory allocated for the Receiver object. This working memory must be `nn::rdt::Receiver::RECEIVER_WORKBUF_ALIGNMENT` (8-byte) aligned, and it must be at least as big as `nn::rdt::Receiver::RECEIVER_WORKBUF_SIZE` (128 bytes).

For the *pRecvBuf* and *recvBufSize* members, specify the starting address and size of the

receive buffer.

For the *nodeId* and *port* members, specify the node ID and port number for the Sender system. These must be the same values as those specified for the UDS library.

Do not free the working memory and receive buffer passed to `Initialize` until you call `Finalize()`.

## 3.4.2. Getting the State

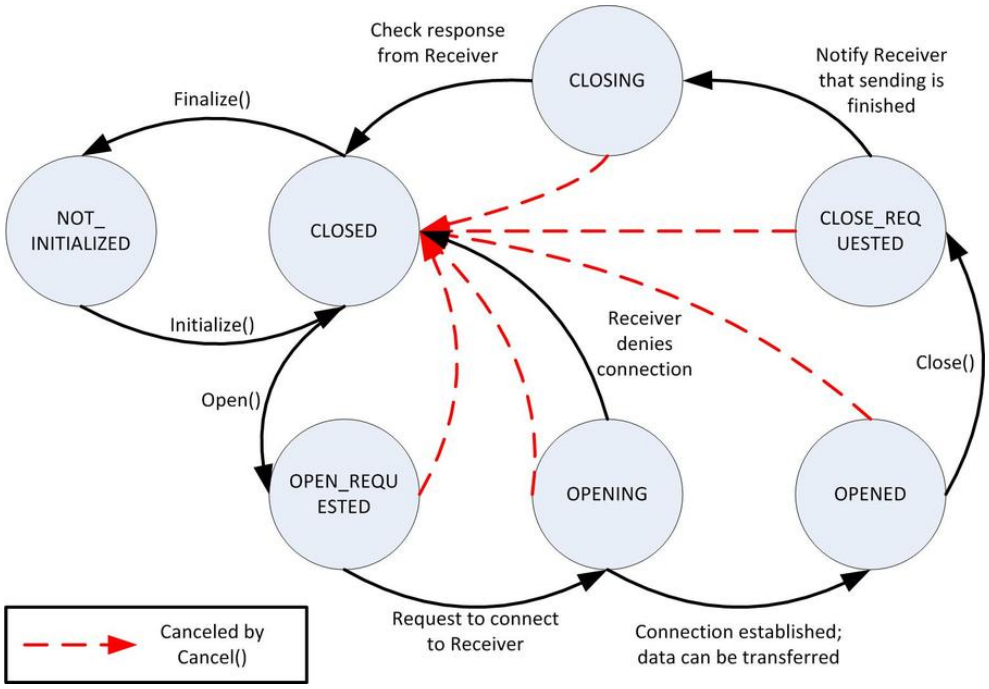
The `Sender` and `Receiver` objects both have internal states. Call the `GetStatus()` member function included in both classes to get their state.

**Code 3-54. Getting the State of Sender and Receiver Objects**

```
enum nn::rdt::SenderState nn::rdt::Sender::GetStatus(void) const;
enum nn::rdt::ReceiverState nn::rdt::Receiver::GetStatus(void) const;
```

The following figure illustrates typical state changes for `Sender` and `Receiver` objects.

**Figure 3-5. Sender State Changes**



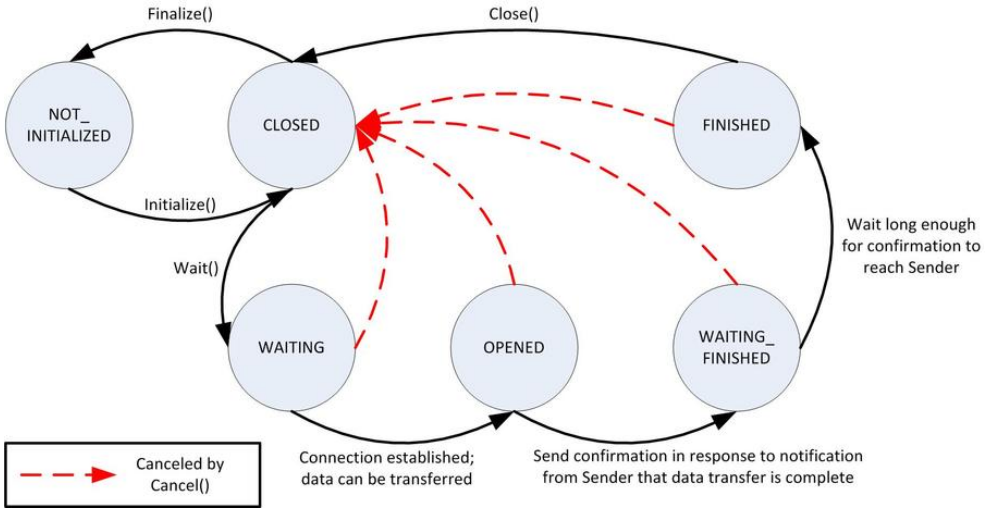

The `Sender` states are defined by `nn::rdt::SenderState`. These states are all prepended with `SENDER_STATE_`.

**Table 3-25. Sender States Defined by nn::rdt::SenderState**

| Definition | Description |
|---|---|
| `SENDER_STATE_NOT_INITIALIZED` | Not yet initialized, such as right after instantiation. |

| | |
|---|---|
| SENDER_STATE_CLOSED | Initialized state. |
| SENDER_STATE_OPEN_REQUESTED | Right after a connection is started. |
| SENDER_STATE_OPENING | Connection request sent and awaiting response. |
| SENDER_STATE_OPENED | Connection established, and data can be transferred. |
| SENDER_STATE_CLOSE_REQUESTED | Right after requesting to close a connection with the Close() function. |
| SENDER_STATE_CLOSING | Sending is complete, and receiver has been notified. |

**Figure 3-6. Receiver State Changes**




The Receiver states are defined by nn::rdt::ReceiverState. These states are all prepended with RECEIVER_STATE_.

**Table 3-26. Receiver States Defined by nn::rdt::ReceiverState**

| Definition | Description |
|---|---|
| RECEIVER_STATE_NOT_INITIALIZED | Not yet initialized, such as right after instantiation. |
| RECEIVER_STATE_CLOSED | Initialized state. |
| RECEIVER_STATE_WAITING | Waiting for connection request. |
| RECEIVER_STATE_OPENED | Connection established, and data can be transferred. |
| RECEIVER_STATE_WAITING_FINISHED | Received notification that data transmission is ending, and sent confirmation response. |
| RECEIVER_STATE_FINISHED | Waited long enough for confirmation to reach the Sender. |

The Receiver changes to the CLOSED state when a connection is marked as closed, such as by a call to the Cancel() function described later in this document.

The individual RDT functions can only be called in some states. Calling a function when the object is in an incompatible state results in an immediate nn::rdt::ResultUntimelyFunctionCall error.

## 3.4.3. Processing Communication

Call the `Process()` member function for both the `Sender` and `Receiver` classes to prompt the `rdt` library to proceed with communication. This function handles data transfer, confirmation, and other RDT communication processes. After initializing the `Sender` and `Receiver` objects, we recommend that the application call `Process()` at least once every frame (at 60 fps, roughly every 16.667 ms).

**Code 3-55. Processing Communication**

```
nn::Result nn::rdt::Sender::Process(void);
 nn::Result nn::rdt::Receiver::Process(void);
```

> **Note:** Although `Sender::Process()` may return `nn::uds::ResultBufferIsFull`, you can safely ignore this error. If the arrival of data cannot be confirmed, that data is automatically resent.

## 3.4.4. Opening a Connection

Call the `Open()` member function of a `Sender` object for that object to attempt to open a connection with the node ID and port number specified for the `Receiver` in the initialization parameters. If the `Receiver` system has called the `Wait()` member function and is in the `WAITING` state, it accepts the connection request, both the `Sender` and `Receiver` transition to the `OPENED` state, and the `Sender` can send data.

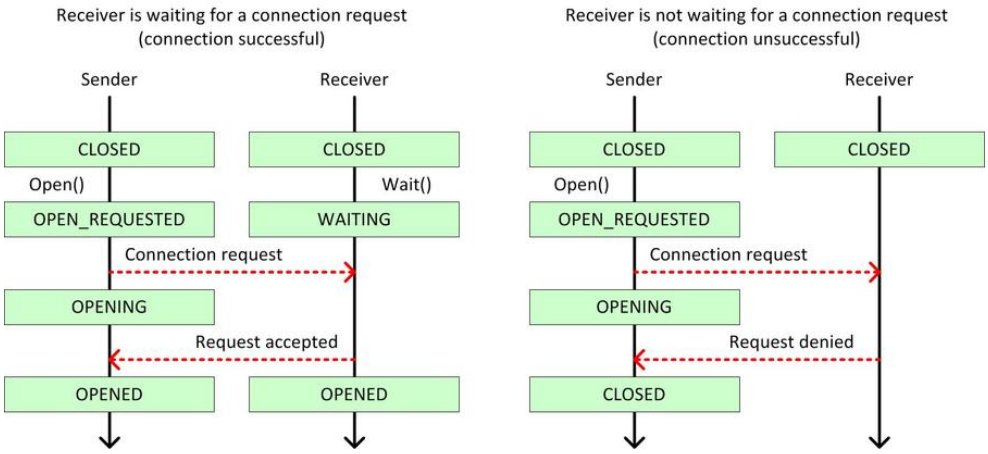
If the `Receiver` is in the `CLOSED` state when the `Sender` requests a connection, the `Receiver` replies with a reset signal to deny the connection request. The `Sender` then transitions to the `CLOSED` state after it gets this signal. Note that this might just mean that the `Receiver` has not called `Wait()` yet, so make sure that you request a connection again if refused.

**Code 3-56. Opening a Connection**

```
nn::Result nn::rdt::Sender::Open(void);
nn::Result nn::rdt::Receiver::Wait(void);
```

The following figure shows state transitions for both the `Sender` and `Receiver` when requesting a connection.

**Figure 3-7. State Changes When Requesting a Connection**




## 3.4.5. Transferring Data

Data can be transferred after both the `Sender` and `Receiver` are in the `OPENED` state.

Call `nn::rdt::Sender::Send()` to send data, and call `nn::rdt::Receiver::Receive()` to receive it. The `Send()` function simply writes data to the send buffer within the `Sender` object, and the `Receive()` function just reads data from the receive buffer within the `Receiver` object. Call the previously mentioned `Process()` function to carry out the raw data sending and receiving.

There are no byte boundaries when sending and receiving data via RDT communication. There is no guarantee that data sent in 1024-byte chunks will arrive in 1024-byte chunks.

> **Note:** Call the `nn::rdt::Sender::Send()` function on the sender side to send data in the order in which it was written to the send buffer. Call the `nn::rdt::Receiver::Receive()` function on the receiver side to receive data.

**Code 3-57. Transferring Data**

```
nn::Result nn::rdt::Sender::Send(const void* pBuf, size_t bufSize);
nn::Result nn::rdt::Receiver::Receive(
                        void* pBuf, size_t* pRecvSize, size_t bufSize);
```

## 3.4.6. Canceling

If the user cancels communication or the UDS library detects a network disconnection, call the `Cancel()` member function for both the `Sender` and `Receiver` classes to stop RDT communication. The `Cancel()` function forces the object into the `CLOSED` state and sends a reset signal to the connection peer. Upon receiving this reset signal, the connection peer also changes to the `CLOSED` state, stopping communication.

**Code 3-58. Canceling**

```
void nn::rdt::Sender::Cancel(void);
void nn::rdt::Receiver::Cancel(void);
```

## 3.4.7. Closing a Connection

Call `nn::rdt::Sender::Close()` after the `Sender` has finished sending data. The `Close()` function notifies the `Sender` that there is no more data to send. The `Sender` object changes to the `CLOSE_REQUESTED` state after `Close()` is called. Calling `Process()` after this transition sends a close request to the `Receiver` and prompts the `Sender` to change to the `CLOSING` state.

The `Receiver` changes to the `WAITING_FINISHED` state after it receives the close request. Calling `Process()` after this transition sends a confirmation response to the `Sender`, and (after waiting long enough for this to reach the `Sender`) the `Receiver` object changes to the `FINISHED` state. At this point, call `Receive()` again to make sure there is no data left to receive (in other words, the receive buffer is empty), and then call `nn::rdt::Receiver::Close()`. The `Receiver` object now changes to the `CLOSED` state and the connection is closed.

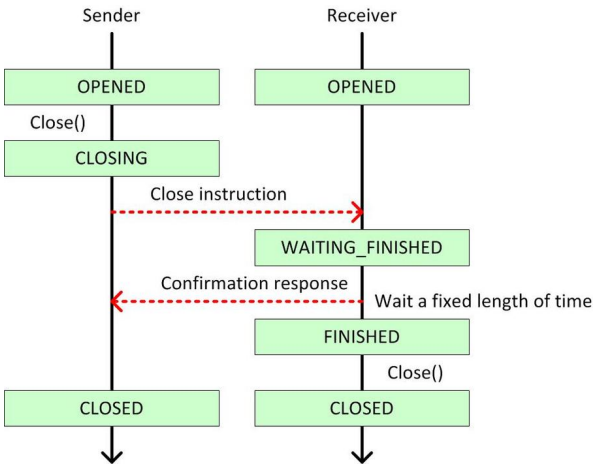
Do not call `Close()` to stop communication. Only call the `nn::rdt::Sender::Close()` function to notify the `Sender` object that all data has been sent. Only call the `nn::rdt::Receiver::Close()` function after making sure there is no data left to receive.

**Code 3-59. Closing a Connection**

```
nn::Result nn::rdt::Sender::Close(void);
nn::Result nn::rdt::Receiver::Close(void);
```

The following figure shows state transitions for both the `Sender` and `Receiver` when closing a connection.

**Figure 3-8. State Changes When Closing a Connection**



### 3.4.8. Finalizing

Call the `Finalize()` member function for both the `Sender` and `Receiver` classes to finalize the objects after you finish using the RDT library. `Finalize()` frees the `nn::uds::EndpointDescriptor` objects implicitly allocated by the instances created on initialization, removing the need to allocate the memory that was passed at initialization.

**Code 3-60. Finalization**

```
void nn::rdt::Sender::Finalize(void);
void nn::rdt::Receiver::Finalize(void);
```



# 4. Background Communication

The following library allows your application to automatically carry out wireless communication (background communication) in the background.

- StreetPass (CEC)
- SpotPass (BOSS)
- Presence feature (FRIENDS)

This chapter describes programming procedures and other information necessary to develop applications using each of these libraries.

#### Effect on Applications of Daemons Running in the Background

The term *daemon* refers to a resident module running in the background. 3DS includes several daemons performing network functions. The network daemon manager (`ndm`) serves as a special daemon that integrates these daemons. An NDM library for controlling `ndm` has been provided in the SDK.

Because the daemons run in a different CPU core than the one in which the application runs, the daemons never directly take up any application CPU time. However, because the application and the daemons both compete for network (wireless communication) access and NAND access, daemon execution does sometimes affect application performance. Applications can stop daemon execution using the NDM library if network and NAND access performance drops to a fatal level.

**Note:** SpotPass stops while an application is running unless explicitly instructed otherwise. This

does not affect either tasks while sleeping or immediate task execution.

Although daemons can be stopped freely, each plays a specific role and stopping one may limit overall operations. Before you stop a daemon, be sure you understand the possible consequences.

The following table shows the currently known effects of daemons on applications.

**Table 4-1. Currently Known Effects of Daemons on Applications**

| Daemon(nn::ndm::DaemonName) | Effect on the Network | Effect on NAND |
|---|---|---|
| nn::ndm::DN_CEC | None. (This daemon does not use infrastructure communication.) | Medium. (When a StreetPass connection is established, there is about 10 KB to 10 MB of NAND access.) |
| nn::ndm::DN_BOSS | High. (HTTP communications occur when the task executes.) | High. (Data is written to NAND during task execution.) |
| nn::ndm::DN_NIM | High. (HTTP communications occur when the task executes.) | High. (Data is written to NAND during task execution.) |
| nn::ndm::DN_FRIENDS | Low. (UDP communications occur when disconnecting.) | Low |

**Warning:** When controlling daemons, make sure that you also read the information in the *CTR-SDK API Reference*.

## 4.1. StreetPass

StreetPass communication on the 3DS system represents improvements in features and ease of use over Chance Encounter Communication on the Nintendo DS.

Communication is performed in the background, allowing you to transmit data by simply passing the data without regard to the state of the running application. However, because this also means that an application cannot implement its own wireless communication processes, it may not be possible to build in certain specialized data transmission methods.

**Warning:** StreetPass requires that a EULA (End-User License Agreement) be set. To use the CEC library, set the **Agree Version** to a value of 1.0 or greater from **EULA Settings** in the `Config` tool.

Communication via StreetPass cannot be carried out between a development CTR system (debugger or development system) and a retail CTR system.

# 4.1.1. Overview of Operations

When using StreetPass communication, the communication processes are handled by the StreetPass daemon (a resident process that runs in the background). StreetPass communication can occur if a StreetPass box has been created in the memory region set aside for the 3DS system and if StreetPass data exists in a StreetPass box. The StreetPass daemon automatically handles sending and receiving StreetPass data. The daemon encrypts data before sending it, so there is no need for individual applications to encrypt data.

There are 12 StreetPass boxes available in the system, with one application generally using one box exclusively. You must get the user's approval before claiming one of the 12 StreetPass boxes to register data for StreetPass. You must get the user's approval before claiming one of the 12 StreetPass boxes to register data for StreetPass. If all 12 StreetPass box memory slots are already in use, inform the user that the application cannot create its StreetPass settings, and that they must go to the StreetPass management screen in System Settings and delete unneeded StreetPass settings for another application. After the user has opened up at least one of the 12 available memory slots for StreetPass boxes, the application can create a new StreetPass box.

**Warning:** When shared use of one system is being made for multiple 3DS Game Cards of the same title, you must be careful with StreetPass that all processes are performed on the same StreetPass box.

Go to the StreetPass management screen in System Settings to check which applications have StreetPass boxes and delete boxes as needed. Deleting a StreetPass box also deletes any notifications in the notification list for that StreetPass box. You must implement your application such that it can continue running even if any data saved in a StreetPass box is deleted.

Within the StreetPass box, there is an inbox and an outbox, each allocated for received data and data to be transmitted.

When a CTR system with StreetPass activated first finds another system in the vicinity that is also activated, the StreetPass boxes in each system's system memory are scanned. If the two systems both have a box created by the same application (in other words, the StreetPass IDs match), each system begins transmitting only the StreetPass data to be received or sent. If the StreetPass IDs are the same, communication is performed even if the system regions differ. If StreetPass data is being sent from multiple StreetPass boxes, data items are sent (according to size) starting with the smallest. The two systems will not necessarily stay within range of each other long enough to finish sending and receiving all StreetPass data. If the systems are within range for a short period of time, only the smallest data item may be successfully sent. Consequently, keep the size of the StreetPass data as small as possible to increase the chance of successful transmission. StreetPass data items that are part of a group are not sent individually.

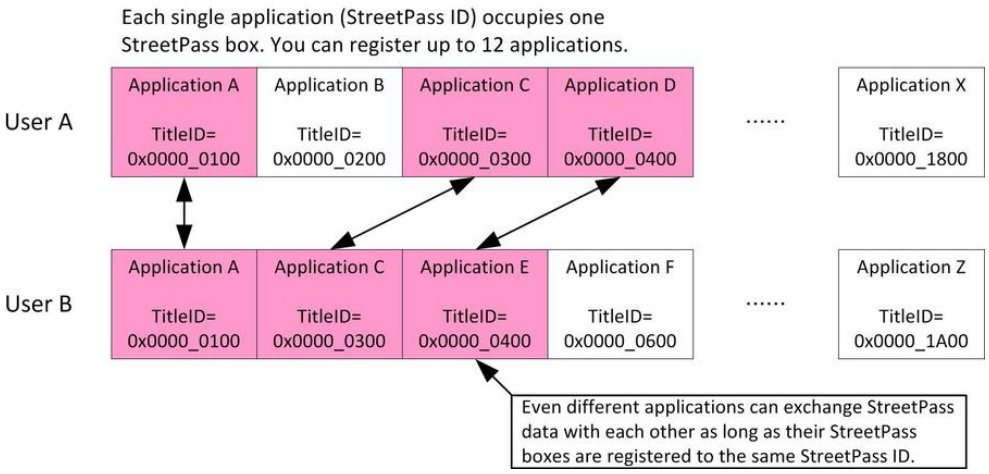
You can specify that StreetPass data be receive-only, but note that two systems do not communicate via StreetPass if the boxes are both set to receive-only or send-only. This situation also applies to units configured to only send StreetPass data. Specifically, if the first registered data is receive-only or send-only, StreetPass exchange never occurs, no matter how much time passes.

The address used by a system for StreetPass communication is regenerated every eight hours and stored in the system's address list. As a result, the interval for StreetPass communication between known communication partners averages four hours and may be up to eight hours. On a development system, you can clear the address list by calling the function to start StreetPass. A similar mechanism to filter addresses cannot be run on a commercial system, so data changes you make may not immediately take effect.

**Note:** StreetPass communication will not take place if the system is connected to an access point registered in the connection settings or to a Nintendo Zone in the vicinity. To check that StreetPass communication is occurring while an application is running, cancel the connection settings and use an environment where there is no nearby Nintendo Zone.

Even when connected to an access point or a Nintendo Zone, StreetPass communication takes place when you switch to StreetPass-exclusive mode (4.1.3.7. StreetPass-Exclusive Mode (for Debugging Only)) or put the system into Sleep Mode.

**Figure 4-1. Scanning StreetPass Boxes**


Each single application (StreetPass ID) occupies one StreetPass box. You can register up to 12 applications.
User A
Application A TitleID= 0x0000_0100
Application B TitleID= 0x0000_0200
Application C TitleID= 0x0000_0300
Application D TitleID= 0x0000_0400
.....
Application X TitleID= 0x0000_1800
User B
Application A TitleID= 0x0000_0100
Application C TitleID= 0x0000_0300
Application E TitleID= 0x0000_0400
Application F TitleID= 0x0000_0600
.....
Application Z TitleID= 0x0000_1A00
Even different applications can exchange StreetPass data with each other as long as their StreetPass boxes are registered to the same StreetPass ID.

## 4.1.1.1. Conditions Related to StreetPass Communication

This section covers conditions for starting StreetPass communication, connecting with communication peers, sending StreetPass data, destroying StreetPass data, and turning on the notification LED.

### Conditions for Starting StreetPass Communication

A search will not be made for peers and StreetPass communication will not start unless all of the following conditions are met.

- The system is in wireless-enabled mode.

- StreetPass communication has not been restricted by Parental Controls.
- The user has agreed to the EULA.
- A StreetPass box has been set up and there is StreetPass data with a possible send count of 1 or more.

Because the StreetPass daemon also searches for Nintendo Zone, daemon status may show as "scanning" even if the all of these conditions are not met.

### Conditions for Connecting with Peers

Connection with a peer cannot be made unless all of the following conditions are met when two systems capable of starting StreetPass communication approach each other.

- The peer has a StreetPass box with the same StreetPass ID.
- The StreetPass data for both systems as arranged to be processed combine to form send/receive modes that allow communication.
- The address used for StreetPass communication with the peer has not been registered in the address filter (a list of addresses of peers with whom StreetPass communication has been performed).
- The peer has not been registered on the blocked-user list.

Addresses for StreetPass communication are regenerated every eight hours. Up to 255 such addresses can be registered in the address filter saved on a system. If a 256th address is registered, the earliest address in the filter is overwritten. In that case, regardless of the status of the StreetPass box, communication with a peer having the overwritten StreetPass address is impossible. StreetPass communication with this same peer remains impossible for an average of four hours and up to eight hours, even if you create a new StreetPass box.

### Conditions for Sending StreetPass Data

After connection with a peer is established, the local host will not send StreetPass data to the peer unless data is exchanged beforehand and all of the following conditions have been met.

- The peer is a friend and communication settings include a flag for sending data only to friends. Or, the peer is not a friend and communication settings include a flag for sending data to peers other than friends.
- When StreetPass data has been sent, the data size and count in the peer’s inbox must not exceed the maximum limits allowed.

Even if these conditions are not met, if the send/receive mode is receive-only or send/receive, receipt of StreetPass data from the peer starts as long as the peer meets these conditions.

If send/receive with the peer cannot start because conditions are not met, the address used for StreetPass communication is registered in the address filter at this point. However, the address is

not registered if the initial exchange of data conducted beforehand failed.

After data transmission has started, the address for StreetPass communication is registered in the address filter if transmission succeeds. The address is not registered if transmission fails.

#### Conditions for Destroying Data Received via StreetPass

If either of the following conditions is met, received StreetPass data is destroyed.

- The StreetPass data signature is incorrect.
- StreetPass data having the same message ID already exists in the inbox.

#### Conditions for Turning on the Notification LED

The notification LED illuminates green if StreetPass data has been received in one of the StreetPass boxes and has not been destroyed.

## 4.1.2. Initialization and Finalization

The application uses the CEC library for StreetPass. Initialize and finalize this library using the `nn::cec::Initialize()` and `nn::cec::Finalize()` functions.

**Code 4-1. Initializing and Finalizing**

```
static nn::Result nn::cec::Initialize(nn::fnd::IAllocator& cecAllocFunc);
nn::Result nn::cec::Finalize();
```

Management of memory that the library uses during StreetPass must be performed by the application's own memory allocator. The library allocates memory from the allocator passed to *`cecAllocFunc`* of approximately twice the maximum size of the StreetPass data, for data processing and other purposes.

Calling `Initialize` after already successfully initializing a StreetPass box returns an `nn::cec::MakeResultAlreadyInitialized` error indicating that the box is already initialized.

## 4.1.3. StreetPass Box

Access StreetPass boxes via the `nn::cec::MessageBox` class.

## 4.1.3.1. Start Accessing (Opening and Creating)

`nn::cec::CTR::MessageBox::CanCreateMessageBox()` is used to test if the StreetPass box can be created before accessing the StreetPass box. Use `nn::cec::MessageBox::CreateMessageBox()` to create the StreetPass box when it is possible to create the box. A StreetPass box is opened after it is successfully created. When a StreetPass box for the target game title already exists, `nn::cec::MessageBox::OpenMessageBox()` is used to open the box and begin access. A new StreetPass box may not be created if the number of registered StreetPass boxes has reached the limit.

When a StreetPass box is opened, the StreetPass daemon stops executing. This status continues until the StreetPass box is closed. If the HOME Menu is displayed while a StreetPass box is open, communication via StreetPass does not occur because the daemon is in a stopped state. If the system enters Sleep Mode while the StreetPass box is open, the daemon resumes operations and returns to stopped status when the system wakes up. Even if StreetPass box contents are updated through StreetPass communication carried out during sleep, updated information is not applied to the opened `MessageBox` class. The box must be reopened to get the latest information. Because the result of accessing the StreetPass box is rolled back, there is a possibility that the result of processes for creating a StreetPass box or adding or deleting StreetPass data will be lost. We recommend that you close or commit open StreetPass boxes before displaying the HOME Menu or transitioning to Sleep Mode.

**Note:** The creation of a StreetPass box and addition or deletion of StreetPass data is not applied until the StreetPass box is closed or committed. We recommend that access to StreetPass boxes be carried out as an uninterrupted series of operations from open to close.

**Warning:** To learn whether real-time StreetPass communication has taken place, use `GetCecEvent`. (For more information, see 4.1.6. Notifying on Communication Activation.) However, because communication stops each time the StreetPass box is opened, if the box is opened and closed at regular intervals, background communication does not take place.

**Code 4-2. Opening and Creating a StreetPass Box**

```
nn::cec::TitleId nn::cec::MakeCecTitleId(bit32 id, bit8 variation = 0x0);
nn::Result nn::cec::MessageBox::CanCreateMessageBox(const TitleId cecTitleId);
nn::Result nn::cec::MessageBox::OpenMessageBox(const TitleId cecTitleId,
                                               const u32 privateId);
nn::Result nn::cec::MessageBox::CreateMessageBox(
        const TitleId cecTitleId, const u32 privateId, const char* hmacKey,
        const void* icon, size_t iconSize, const wchar_t* name, size_t
nameSize,
        size_t inboxSizeMax = CEC_INBOX_SIZE_DEFAULT,
        size_t outboxSizeMax = CEC_OUTBOX_SIZE_DEFAULT,
        size_t inboxMessNumMax = CEC_INBOX_MESSNUM_DEFAULT,
        size_t outboxMessNumMax = CEC_OUTBOX_MESSNUM_DEFAULT,
```

```
            size_t messageSizeMax = CEC_MESSSIZEMAX_DEFAULT);
```

*cecTitleId* specifies the Nintendo-assigned value for identifying a game title (the StreetPass ID). The StreetPass ID is generated by calling `nn::cec::MakeCecTitleId` with a unique 20-bit ID as an argument. This 20-bit ID is assigned by Nintendo. To share StreetPass with multiple titles, specify one of the unique StreetPass IDs as a representative value. The titles will share a single StreetPass box.

*privateId* specifies the key value required to access a StreetPass box. Developers may choose this value freely. Depending on how this value is generated, access may be restricted to specific save data or to specific game cards.

A StreetPass box may only be opened if these two parameters specify the same values used when creating that StreetPass box. A "no data" error is returned if no StreetPass box exists for the specified StreetPass ID in the *cecTitleId* value, and a "no access permissions" error is returned if a StreetPass box exists but the *privateId* value differs.

*hmacKey* specifies a 32-byte string used to prevent spoofing. Developers may choose this string freely. This value must be identical for both the sending and receiving units. In other words, applications using StreetPass for the same StreetPass ID must specify the same string value.

For the *icon* parameter, specify the icon to display on the StreetPass management screen, and for *iconSize*, specify the icon's size. Icons must be 48×48 pixels in size using the RGB565 PICA native pixel format. You may usually use the game title icon, but for titles in a series, you may use a different icon that makes it clear that this title is part of that series.

For the *name* parameter, specify the title to display on the StreetPass management screen, and for *nameSize*, specify its length in bytes. Titles must be strings of up to 64 characters, including the terminating character. However, the actual display length is equivalent to 17 characters of the system fonts at maximum width (in Japan, European, U.S. regions: "%"; in other regions: Japanese hiragana characters, Chinese kanji characters, Hangul characters, and so on). The language of the title can be the same as the language configured in the CTR System settings, or as configured in the language settings of the application itself.

The *inboxSizeMax* and *outboxSizeMax* parameters specify the maximum size of the inbox and outbox. The total size of both boxes cannot exceed 1 MB. Unless propagating StreetPass data, the basic rule is to allocate a substantial size for the inbox. If no values are specified for these parameters, the default size is 512 KB.

The *inboxMessNumMax* and *outboxMessNumMax* parameters specify the maximum number of messages that can be saved to the inbox and outbox. Specify the value derived by dividing the box capacity by the size of the largest StreetPass message. If no values are specified for these parameters, the default value is 99.

*messageSizeMax* specifies the maximum size of a StreetPass message. No StreetPass data larger than the specified value may be saved. In general, omit this argument to use the default value of 100 KB. The maximum size of StreetPass data is fixed at 100 KB on the current CTR-SDK, regardless of the value specified for *messageSizeMax*.

The following table lists errors that can be returned by the `nn::cec::MessageBox::OpenMessageBox()` function, and how to resolve them.

**Table 4-2. Errors That May Be Returned by the OpenMessageBox Function and Their Resolution**

| Error | Cause and Resolution |
|---|---|
| `nn::cec::ResultNoData` | No StreetPass box exists. Create a StreetPass box and reopen. |
| `nn::cec::ResultNotAuthorized` | The StreetPass box cannot be opened because the *`privateId`* values do not match. Data must be deleted when re-creating the StreetPass box. |
| `nn::cec::ResultStateBusy` | This error is not returned unless there has been access from multiple threads. The StreetPass box can be opened by waiting awhile and retrying. |
| Any other errors | If an error not in this table is returned, function execution is treated as having failed. The application must be allowed to advance without using this function. If this function is required for the application to advance, this error may be displayed as a fatal error. |

The following table lists errors that can be returned by the `nn::cec::MessageBox::CreateMessageBox()` function, and how to resolve them.

**Table 4-3. Errors That May Be Returned by the CreateMessageBox Function and Their Resolution**

| Error | Cause and Resolution |
|---|---|
| `nn::cec::MakeResultBoxAlreadyExists` | A StreetPass box having the same *`cecTitleId`* and *`privateId`* has already been created.<br>If the `nn::cec::ResultNoData` error is returned by the `nn::cec::MessageBox::OpenMessageBox()` function, there is already a StreetPass box that differs only in terms of its *`cecTitleId`*. |
| `nn::cec::MakeResultBoxNumFull` | There are already twelve StreetPass boxes. A new box cannot be created. Instruct the user to delete an unneeded StreetPass box on the StreetPass Management screen. |
| `nn::cec::ResultStateBusy` | This error is not returned unless there has been access from multiple threads. The StreetPass box can be opened by waiting awhile and retrying. |
| `nn::cec::ResultInvalidArgument` | An argument has been specified incorrectly. |
| `nn::cec::ResultTooLarge`<br>`nn::cec::ResultInvalidData` | An icon or name has been specified incorrectly. |
| Any other errors | If an error not in this table is returned, function execution is treated as having failed. The application must be allowed to advance without using this function. If this function is required for the application to advance, this error may be displayed as a fatal error. |

## 4.1.3.2. Setting Related Data

You can change the icon and title displayed on the StreetPass Management screen. Open the StreetPass box and set the new icon and title as related data.

**Code 4-3. Setting Related Data**

```
nn::Result nn::cec::MessageBox::SetMessageBoxData(
                        u32 datatype, const void* data, size_t dataSize);
```

*datatype* specifies the type of related data. If setting an icon, use the value `nn::cec::BOXDATA_TYPE_ICON`, and if setting the title, use the value `nn::cec::BOXDATA_TYPE_NAME_1`. Specify the starting address of the related data and its size in the *data* and *dataSize* parameters.

Icons must be 48×48 pixels in size using the RGB565 PICA native pixel format. You may usually use the game title icon, but for titles in a series, you may use a different icon that makes it clear that this title is part of that series.

Titles must be strings of up to 64 characters, including the string terminator. The actual display length is equivalent to 17 characters of the system fonts at maximum width (in Japan, European, U.S. regions: "%"; in other regions: Japanese hiragana characters, Chinese kanji characters, Hangul characters, and so on). The language of the title can be the same as the language configured in the CTR System settings, or as configured in the language settings of the application itself.

## 4.1.3.3. Inbox and Outbox Information

You can get information about the inbox and outbox, such as the capacity or maximum number of messages specified at box creation, the currently used capacity, or the number of messages currently saved.

**Code 4-4. Getting Inbox and Outbox Information**

```
u32 nn::cec::MessageBox::GetBoxSizeMax(nn::cec::CecBoxType boxType) const;
u32 nn::cec::MessageBox::GetBoxSize(nn::cec::CecBoxType boxType) const;
u32 nn::cec::MessageBox::GetBoxMessageNumMax(nn::cec::CecBoxType boxType)
const;
u32 nn::cec::MessageBox::GetBoxMessageNum(nn::cec::CecBoxType boxType) const;
```

All of these functions can be used for either the inbox or outbox, specifying which one by passing an `nn::cec::CecBoxType` enumerator as an argument. Specify `CEC_BOXTYPE_INBOX` for the inbox and `CEC_BOXTYPE_OUTBOX` for the outbox.

Use the `GetBoxSizeMax()` and `GetBoxSize()` functions to get the capacity and used capacity of the specified box.

Use the `GetBoxMessageNumMax()` and `GetBoxMessageNum()` functions to get the maximum number of StreetPass messages and the current number of saved messages for the specified box.

You can scan the data in a StreetPass box by constructing a loop over the number of messages currently saved in it, using the `nn::cec::MessageBox` class's `GetMessageId()` functions to get the message ID corresponding to the index.

**Code 4-5. Getting a Message ID by Specifying an Index**

```
nn::Result nn::cec::MessageBox::GetMessageId(MessageId* messId,
                        const CecBoxType boxType, const u32 messIndex);
```

Call the `ReadMessage()` function passing the obtained message ID as an argument to get a StreetPass message, and then access that message via the `nn::cec::Message` class.

**Code 4-6. Getting a StreetPass Message**

```
u32 nn::cec::MessageBox::GetMessageSize(const CecBoxType boxType,
                                        const u32 messIndex) const;
nn::Result nn::cec::MessageBox::ReadMessage(
                nn::cec::Message& cecMessage, void* buf, const size_t bufLen,
                const CecBoxType boxType, const MessageId& messageId);
```

*cecMessage* takes an instance of the `nn::cec::Message` class.

The *buf* and *bufLen* parameters must specify a buffer of the size returned by a call to `GetMessageSize`, and the size, respectively. This is because the message is first read as a binary array, and then the information is set in the class instance passed in *cecMessage*.

For more information about accessing StreetPass data via the `nn::cec::Message` class, see 4.1.4.3. Getting Information.

## 4.1.3.4. Deleting

StreetPass boxes are a limited resource with only 12 available on the system. Be sure to delete boxes that contain no StreetPass data, or when users are not engaging in StreetPass to allow other applications to create StreetPass boxes.

**Code 4-7. Deleting a StreetPass Box**

```
nn::Result nn::cec::MessageBox::DeleteMessageBox();
nn::Result nn::cec::MessageBox::DeleteMessageBox(const TitleId cecTitleId);
```

*cecTitleId* specifies the StreetPass ID of the StreetPass box to delete. If this parameter is omitted, the box currently opened by the `nn::cec::MessageBox` class is deleted.

**Warning:** Do not delete StreetPass boxes other than created by the application.

Only delete StreetPass boxes with the specified StreetPass ID that can no longer be opened because they are broken.

## 4.1.3.5. Ending Access (Close and Commit)

After access to a StreetPass box has ended, the daemon must be returned to executing status. To restore execution status, call `nn::cec::MessageBox::CloseMessageBox` to close an open StreetPass box.

**Code 4-8. Closing a StreetPass Box**

```
void nn::cec::MessageBox::CloseMessageBox();
```

Changes to a StreetPass box are not written into save memory until they are committed. If an application exits before committing this data, content may rewind to data committed in the past. Call `nn::cec::MessageBox::CommitMessageBox` to commit, but not that it is also committed when closing the StreetPass box.

**Code 4-9. Committing a StreetPass Box**

```
nn::Result nn::cec::MessageBox::CommitMessageBox();
```

## 4.1.3.6. Relationship With the StreetPass Daemon

When a StreetPass box is opened by an application, the StreetPass daemon stops execution. If this happens during the execution of a daemon process, the process is canceled and any received data may be lost completely. StreetPass boxes can be safely accessed by using the `nn::cec::CecControl::GetCecState()` function to check if the daemon is busy (`DAEMON_STATE_BUSY`) or communicating (`DAEMON_STATE_COMMUNICATING`) before opening a StreetPass box.

## 4.1.3.7. StreetPass-Exclusive Mode (for Debugging Only)

You can set the system to use only StreetPass for background communication by using the following functions. Note that these are only to be used for debugging purposes.

**Code 4-10. Switching to StreetPass-Exclusive Mode**

```
static nn::Result nn::cec::CecControl::EnterExclusiveState();
static nn::Result nn::cec::CecControl::LeaveExclusiveState();
```

These functions produce an error when switching to StreetPass-exclusive mode if the system is either already using exclusive background communication or is carrying out local communication.

## 4.1.4. StreetPass Data

All access to StreetPass data is carried out via the `nn::cec::Message` class.

### 4.1.4.1. Creating New Data

To create new StreetPass data, prepare an instance of the `nn::cec::Message` class and call its `NewMessage` member function.

**Code 4-11. Creating New StreetPass Data**

```
nn::Result nn::cec::Message::NewMessage(TitleId cecTitleId, u32 groupId,
                        u8 messageTypeFlag, u8 sendMode,
                        u8 sendCount, u8 propagationCount);
nn::Result nn::cec::Message::NewMessage(TitleId cecTitleId, u32 groupId,
                        u8 messageTypeFlag, u8 sendMode,
                        u8 sendCount, u8 propagationCount,
                        const void* icon, size_t iconSize,
                        const wchar_t* infoTextData, size_t infoTextSize);
```

*`cecTitleId`* specifies the StreetPass ID specified when the StreetPass box was opened.

*`groupId`* specifies the group number when grouping StreetPass data. Note that the total size for a single group cannot exceed the maximum size for a single transmission (100 KB). Specify 0 for this parameter to not group data and to send messages individually, even if a data group with a number of 0 actually exists.

*`messageTypeFlag`* specifies a bitwise OR of flags indicating the recipients of the data. Using a combination of flags, you can restrict the StreetPass data to be sent only to friends or only to non-friends, or allow it to be sent to both. This setting makes it possible to send different StreetPass messages to friends and non-friends and, by registering StreetPass data with different flags in the same group, send additional comments to friends or otherwise control how data is sent. These flags are only considered when data is sent. You cannot make a user's status as a friend (or not) a condition for receiving data.

**Table 4-4. Message Recipient Flags**

| Flag | Description |
|---|---|
| | Set when sending without differentiating between friends and |

| | |
|---|---|
| Bitwise OR of the following two flags | non-friends.<br>This setting is the default, and it sends to everyone. |
| MESSAGE_TYPEFLAG_NON_FRIEND | Sends data only to communication partners who are not friends.<br>Set this flag when sending different comments for friends and non-friends in the StreetPass data for the same group, or in similar situations. |
| MESSAGE_TYPEFLAG_FRIEND | Sends data only to communication partners who are friends.<br>Set this flag when adding comments to be sent just to friends included in the same group as StreetPass data to send to everyone, or in similar situations. |

**Warning:** If you specify only MESSAGE_TYPEFLAG_FRIEND, make sure that the different StreetPass Relay process does not cause any problems for your application's specifications. For more information, see 9. Appendix: StreetPass Relay.

*sendMode* specifies the transmission mode for controlling StreetPass data transmission. The combination of transmission modes for communication partners controls whether StreetPass data is sent and received during communication.

**Table 4-5. Transmission Modes**

| Transmission Mode | Description |
|---|---|
| CEC_SENDMODE_RECV | Receive only. Data is received when the communication partner is in send only or send/receive mode.<br>We recommend not using this setting because it prevents StreetPass communication with peers who are using the same setting.<br>StreetPass data is registered in the outbox. |
| CEC_SENDMODE_SEND | Send only. Data is sent when the communication partner is in receive only or send/receive mode.<br>We recommend not using this setting because it prevents StreetPass communication with peers who are using the same setting. |
| CEC_SENDMODE_SENDRECV | Send/receive. Data is sent when the communication partner is in receive only mode, received when the communication partner is in send only mode, and sent and received when the communication partner is in send/receive mode.<br>Specify this mode for one-way communication (only sending or only receiving data), such as when users distribute their own creations, invitations to in-game houses, or similar situations. |
| CEC_SENDMODE_EXCHANGE | Exchange. Data is sent and received when the communication partner is also in exchange mode. StreetPass communication is not successfully established if either party cannot properly send and receive.<br>Specify this mode when specific data must be exchanged, such as when swapping in-game pets or engaging in StreetPass matches. |

**Table 4-6. Transmission in Each of the Transmission Mode Combinations**

| | | | | |
|---|---|---|---|---|

| A (Row)<br>B (Column) | Receive Only | Send Only | Send/Receive | Exchange |
|---|---|---|---|---|
| Receive Only | × | A ← B | A ← B | × |
| Send Only | A → B | × | A → B | × |
| Send/Receive | A → B | A ← B | A ⇔ B | × |
| Exchange | × | × | × | A ⇔ B |

There is no guarantee that StreetPass data has been successfully sent. Because grouping is performed even if the send/receive mode differs, use the same send/receive mode for data associated with the same group whenever possible, while making sure that the send/receive mode does not include receive only data.

**Warning:** When the transmission mode is Exchange, communications are only carried out between systems that are set to the same Exchange mode. If there is a mix of transmission modes set for the same StreetPass box, the probability of StreetPass occurrences will decline.

*sendCount* specifies the number of times to send a StreetPass message. Specify `nn::cec::MESSAGE_SENDCOUNT_UNLIMITED` to not limit the number of times a message can be sent. The *sendCount* value is decremented by 1 every time a message is sent and sending stops when the counter reaches 0. The sendable count decrements at the point data is sent, regardless of whether data received by the receiver has been registered in the inbox. The value decrements when data is sent, and does not decrement when the communication peer is in send only mode, even if the local system is in send/receive mode. This value will also not decrement when the communication peer's box is full and data cannot be sent, such as after reaching the maximum number of times a message can be received. However, data can still be received from this peer so long as there is room in the local inbox. The *sendCount* value acts as a limit on the number of times data can be received when in receive only mode (`CEC_SENDMOD_RECV`), decrementing by 1 every time data is received until reaching 0.

If the same StreetPass data is received, the newer StreetPass data is destroyed. Even if StreetPass data is destroyed, the *sendCount* of the sender is decremented by one.

There is no guarantee that sent data will actually reach the communication peer. A power loss immediately after StreetPass communication is established and while in the middle of saving data or decrementing the *sendCount* value can cause an irreconcilable inconsistency between the remaining *sendCount* value and the number of data items received. For example, when the connection is broken nearly at the same time as the transmission is completed, the following cases occur based on the specific timing.

- The sender fails to send, and the receiver successfully receives. In this case, the sender's send count does not decrease, but the StreetPass data is received.
- The send side successfully sends, and the receive side fails to receive. In this case, the sender's send count decreases, and the StreetPass data is not received.

**Note:** The address filter will add the address if the device successfully sends or receives at least one message.

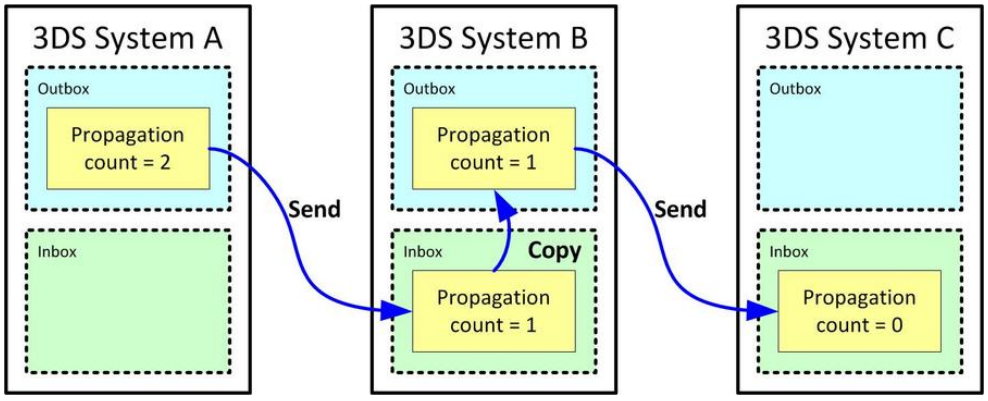
When the connection between two devices that started a StreetPass communication is broken, and one device is successful at sending and fails to receive, and the other device is successful at sending and fails to receive, each device records the other in its address filter, even if there is remaining data that was not sent.

In this case, the address filter may limit communication, creating delay until the devices can resume StreetPass when other title data's StreetPass communication has occurred, and the expected title data's StreetPass communication has not occurred.

*propagationCount* specifies the number of times a StreetPass message is propagated. This value is decremented by 1 every time the message is received, and a message with a propagation value of at least 1 is copied to the outbox after receipt. In other words, specify a value of at least 2 to propagate a StreetPass message. Note that received data can be saved in both the inbox and outbox, and that you may receive StreetPass data that you sent yourself, depending on the route through which it is propagated. The library imposes a restriction that either the *sendCount* or *propagationCount* parameters must be set to 1 to prevent the risk that a single StreetPass message could multiply exponentially.

If StreetPass data for which a propagation count has been set is received, and the same data already exists in the inbox, the data will not be propagated further because the new data is destroyed. Even if the data is received, and the same data does not already exist in the inbox, it will not be propagated if there is not enough capacity in the outbox. However, if the same data has already been received and copied to the outbox, propagation continues to update the propagation count and possible send count (which must be 1) to match the new data.

**Figure 4-2. Propagation With a Propagation Count of 2**




The `nn::cec::NewMessage()` function mainly sets the conditions for sending and receiving a message. However, other information such as the data to display in the notification (icon, text) and the actual data to send must also be set. To specify such additional information at the time of message creation, use the overloaded version of `nn::cec::Message::NewMessage()` and specify values for the *icon*, *iconSize*, *infoTextData*, and *infoTextSize* parameters. The language configured for the description in *infoTextData* can be the same as the language configured in the CTR System settings, or as configured in the language settings of the application itself. But text for only one language can be configured, so if the settings are different

for the sender and receiver, the description could be displayed for the receiver in a language different from the receiver's configured language.

Use the `nn::cec::Message::SetaExHeader()` function to set the data to display in the notification as extended header information.

**Code 4-12. Setting Extended Header Information**

```
nn::Result nn::cec::Message::SetExHeader(
        const u32 exhType, const size_t exhLen, const void* exhBody);
```

*exhType* specifies the type of extended header information.

**Table 4-7. Extended Header Types**

| Type | Description |
|---|---|
| `MESSAGE_EXHEADER_TYPE_ICON` | Icon. Its size must be 40×40 pixels using the RGB565 PICA native pixel format. |
| `MESSAGE_EXHEADER_TYPE_INFO` | Descriptive text. Up to two lines each having a width equal to 16 characters (in Japan, European, U.S. regions: "%"; in other regions: Japanese hiragana characters, Chinese kanji characters, Hangul characters, and so on) of the internal font at maximum width may be displayed. Maximum length of 128 characters, including newlines and the string terminator. Encoded as UTF16-LE. |

The *exhBody* and *exhLen* parameters specify the data for the extended header information and its size.

You can set the icon data by calling the `nn::cec::Message::SetIcon()` function and set the descriptive text by calling the `nn::cec::Message::SetInfoText()` function.

Set the data to send in a StreetPass message by calling the `nn::cec::Message::SetMessageBody()` function.

**Code 4-13. Setting the Data to Send**

```
nn::Result nn::cec::Message::SetMessageBody(
                        const void* dataBody, const size_t size);
```

The *dataBody* and *size* parameters specify the data to send and its size. The size must be a multiple of `nn::cec::Message::MESSAGE_BODY_UNITSIZE` (4).

The total size of the header information, extended header information, and data to send cannot exceed the maximum StreetPass data size specified when creating the StreetPass box (default size of 100 KB). The maximum size of send data is `nn::cec::Message::MESSAGE_BODY_SIZE_MAX` (96 KB). However, the larger the data size, the later a message is positioned in the StreetPass box sending queue and the longer the time required to send it, which may reduce the chances of successfully transmitting the message.

You do not need to configure receive-only StreetPass data with extended header information and data to send.

You can also call the `nn::cec::Message::SetTag()` function to set an optional 16 bits of data. Use this when it is convenient, such as for a short message identifier that would be seen when accessing just a message header without reading the data portion.

## 4.1.4.2. Registering to the Outbox

After a StreetPass message is composed, it must be saved to the outbox using the `WriteMessage` member function of the `nn::cec::MessageBox` class.

**Code 4-14. Saving a StreetPass Message to a Message Box**

```
nn::Result nn::cec::MessageBox::WriteMessage(const nn::cec::Message&
cecMessage,
                          const CecBoxType boxType, MessageId& messageIdOut);
```

*cecMessage* specifies an instance of the `nn::cec::Message` class representing the created StreetPass data.

*boxType* usually specifies the outbox (`CEC_BOXTYPE_OUTBOX`). Applications are not required to perform operations to specify the inbox.

*messageIdOut* specifies the variable that the message ID is written to after the StreetPass data is saved by a call to this function.

A check for the EULA agreement is made when StreetPass data is saved, so the FS library must be initialized beforehand. The error `ResultNotAgreeEula` is returned if the user has not agreed to the EULA, or if an icon file has not been set in the application. The error `ResultParentalControlCec` is returned if StreetPass has been restricted by Parental Controls.

**Note:** The maximum size of StreetPass data is fixed at 100 KB. An excessive size error (`ResultMessTooLarge`) does not occur unless data exceeds 100 KB.

### Order of Registering StreetPass Data in the Outbox and Processing It

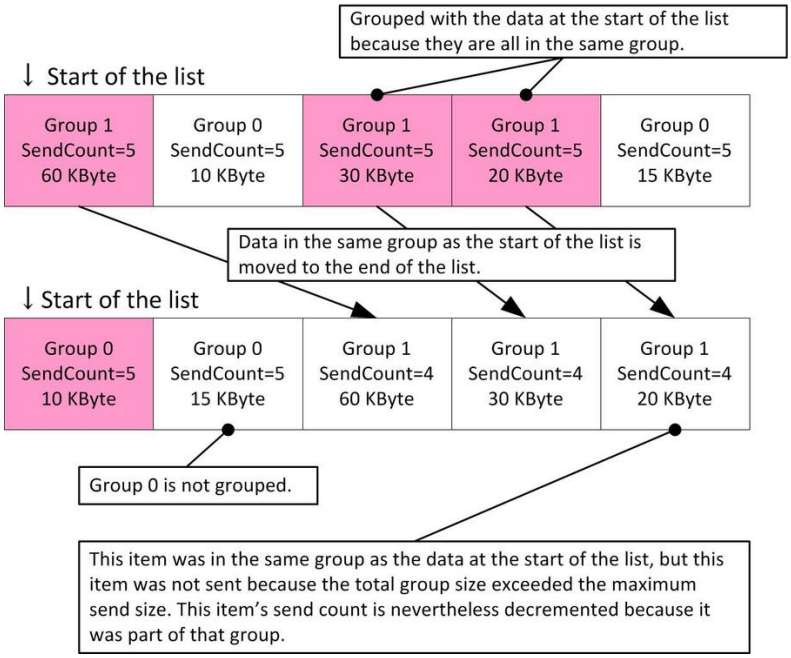
A list is created in a StreetPass box in the same order that data was registered with the outbox. The data at the start of this list is generally processed first unless its allowable send count is 0, in which case the next data in the list (in order) is processed. The send/receive mode of the data to be processed determines whether it is actually sent and received. If data whose send/receive mode has been set to receive only is included in a group, that data is sent to the communicating peer. Be sure to place data so that groups are formed from data having the same send/receive mode.

If a nonzero group ID is set for the data to be processed and there is other data with the same group in the list, that data is processed together until either the end of the list is reached or the total data size reaches the maximum amount that can be sent in a single transmission. The data from the same group as at start is then moved to the end of the list, and the data at the start of the list is replaced. All data (processed or not) with the same group as this processed data have their *sendCount* values decremented by 1.

When the group ID for the data to be processed is set to 0, only that data is processed. Any other data with a group ID of 0 is not treated as part of the same group. Only the processed data is appended to the end of the list, and the *sendCount* value is decremented by 1.

When you register new StreetPass data, it is appended to the end of the list. Any received messages that have *propagationCount* values of at least 1 after decrementing are appended to the end of the list (added to the outbox), with such messages received from other systems then being sent out at the next opportunity for StreetPass communication.

**Figure 4-3. Group Settings and Process Order**


Grouped with the data at the start of the list because they are all in the same group.
Start of the list
Group 1 SendCount=5 60 KByte
Group 0 SendCount=5 10 KByte
Group 1 SendCount=5 30 KByte
Group 1 SendCount=5 20 KByte
Group 0 SendCount=5 15 KByte
Data in the same group as the start of the list is moved to the end of the list.
Start of the list
Group 0 SendCount=5 10 KByte
Group 0 SendCount=5 15 KByte
Group 1 SendCount=4 60 KByte
Group 1 SendCount=4 30 KByte
Group 1 SendCount=4 20 KByte
Group 0 is not grouped.
This item was in the same group as the data at the start of the list, but this item was not sent because the total group size exceeded the maximum send size. This item's send count is nevertheless decremented because it was part of that group.

## 4.1.4.3. Getting Information

Get required information via the `nn::cec::Message` class to access received data obtained by scanning the inbox.

**Code 4-15. Getting Information**

```
u32 nn::cec::Message::GetBodySize() const;
u32 nn::cec::Message::GetMessageBody(void* dataBody, size_t size) const;
MessageId nn::cec::Message::GetMessageId_Pair(MessageId* messIdPair) const;
u16 nn::cec::Message::GetTag() const;
```

Call the GetMessageBody member function to get the sent data in the StreetPass message.

The *dataBody* and *size* parameters specify a buffer to store the sent data and the buffer size. Use the GetBodySize() function to get the required size of the buffer.

If the StreetPass message transmission mode was set to CEC_SENDMODE_EXCHANGE, access data by getting the message ID of the data sent from the communication partner in the GetMessageId_Pair() function.

Use the GetTag() function to get the tag information in the StreetPass message.

## 4.1.4.4. Deleting

Sent or received StreetPass messages are not automatically deleted. The amount of data and number of messages that may be saved in the outbox or inbox is limited. An application must delete StreetPass messages that are no longer needed, such as data to send that has a sendCount value of 0, or received data that has already been applied to the application. Delete messages using the DeleteMessage member function of the nn::cec::MessageBox class. To delete all StreetPass data in the StreetPass box, call the DeleteAllMessages member function of the nn::cec::MessageBox class.

**Code 4-16. Deleting a StreetPass Message**

```
nn::Result nn::cec::MessageBox::DeleteMessage(
                const CecBoxType boxType, const MessageId& messageId);
nn::Result nn::cec::MessageBox::DeleteAllMessages(const CecBoxType boxType);
```

Specify the ID of the message to delete in the *messageId* parameter, and the type of message box where the message is saved in the *boxType* parameter.

If the StreetPass box holds two or more messages, and you want to delete all of them, it is faster to use DeleteAllMessages than to delete them individually using DeleteMessage.

## 4.1.4.5. Updating Information

The information for StreetPass data in the StreetPass box is updated via the nn::cec::Message class.

You read the StreetPass data that is going to be updated using the ReadMessage member function of the nn::cec::MessageBox class, set the attribute values to update with the obtained instance of the nn::cec::Message class, and then update the information by writing to the StreetPass box using the WriteMessage member function of that instance of the nn::cec::MessageBox class.

After the information has been written, call the nn::cec::MessageBox member function CommitMessageBox or CloseMessageBox to commit the updated information to the StreetPass

box.

The following example shows how to use member functions to update StreetPass data attributes.

**Code 4-17. Setting StreetPass Data**

```
nn::Result nn::cec::Message::SetGroupID(u32 groupId);
nn::Result nn::cec::Message::SetMessageTypeFlag(MessageTypeFlag messTypeFlag);
nn::Result nn::cec::Message::SetSendMode(SendMode sendMode);
void nn::cec::Message::SetTag(u16 tag);
```

**Note:** Some of the attributes of the `nn::cec::Message` object obtained from inside the StreetPass box (such as the extended header information) cannot be edited, and when you attempt to update them you will get an `ResultNotAuthorized` error.

For more information about attribute settings, see the API reference.

## 4.1.5. Accessing Header Information

You can use the `nn::cec::MessageBox` class to access just the header information for a StreetPass box or StreetPass message to extract limited information without having to open the data portion. Note that this class cannot be used to access the data to send, nor the extended header information.

Use the following member functions to get the StreetPass message total size and other information.

**Code 4-18. Accessing StreetPass Message Information**

```
u32 nn::cec::MessageBox::GetMessageSize(
                         const CecBoxType boxType, const u32 messIndex) const;
u32 nn::cec::MessageBox::GetMessageBodySize(
                         const CecBoxType boxType, const u32 messIndex) const;
u32 nn::cec::MessageBox::GetMessageGroupId(
                         const CecBoxType boxType, const u32 messIndex) const;
u32 nn::cec::MessageBox::GetMessageSessionId(
                         const CecBoxType boxType, const u32 messIndex) const;
u8 nn::cec::MessageBox::GetMessageTypeFlag(
                         const CecBoxType boxType, const u32 messIndex) const;
u8 nn::cec::MessageBox::GetMessageSendMode(
                         const CecBoxType boxType, const u32 messIndex) const;
u8 nn::cec::MessageBox::GetMessageSendCount(
                         const CecBoxType boxType, const u32 messIndex) const;
u8 nn::cec::MessageBox::GetMessagePropagationCount(
                         const CecBoxType boxType, const u32 messIndex) const;
bit16 nn::cec::MessageBox::GetMessageTag(
                         const CecBoxType boxType, const u32 messIndex) const;
nn::fnd::DateTimeParameters nn::cec::MessageBox::GetMessageSendDate(
                         const CecBoxType boxType, const u32 messIndex) const;
nn::fnd::DateTimeParameters nn::cec::MessageBox::GetMessageRecvDate(
                         const CecBoxType boxType, const u32 messIndex) const;
```

```
nn::fnd::DateTimeParameters nn::cec::MessageBox::GetMessageCreateDate(
                        const CecBoxType boxType, const u32 messIndex) const;
MessageId nn::cec::MessageBox::GetMessageIdPair(
                        const CecBoxType boxType, const u32 messIndex) const;
nn::Result nn::cec::MessageBox::GetMessageIdPair(
        MessageId* messId, const CecBoxType boxType, const u32 messIndex) const;
MessageId nn::cec::MessageBox::GetMessageId(
                        const CecBoxType boxType, const u32 messIndex) const;
nn::Result nn::cec::MessageBox::GetMessageId(
        MessageId* messId, const CecBoxType boxType, const u32 messIndex) const;
u32 nn::cec::MessageBox::GetMessageIndex(
                        CecBoxType boxType, const MessageId& messId) const;
u32 nn::cec::MessageBox::GetMessageIndex(CecBoxType boxType, u8* messId) const;
```

## 4.1.6. Notifying on Communication Activation

You can receive a notification event indicating that the communication has been activated for StreetPass data. You can also get information about the most recent received data.

**Code 4-19. Getting Communication Activation Events and Received Data**

```
nn::Result nn::cec::GetCecEvent(nn::os::Event* event);
nn::Result nn::cec::GetCecInfoBuffer(
                        u32 cecTitleId, u8 pCecInfoBuffer[], size_t size);
```

For the `nn::os::Event` class instance specified in the *`event`* parameter, specify an instance that has not been initialized. Note that this event is signaled when the StreetPass box registered by an application that has obtained the event receives StreetPass data, and also when any StreetPass box registered by any other application receives StreetPass data or when StreetPass data was sent. Use this event for immediate notification on the reception of StreetPass data during background StreetPass communication while an application is running.

Call the `nn::cec::GetCecInfoBuffer()` function to get information about the most recent received data. This function stores information about the last StreetPass communication in the buffer specified by the *`pCecInfoBuffer`* parameter. In general, call this function passing a value for *`cecTitleId`* to determine whether data has been received in the local inbox after detecting communication activation by means of the previously mentioned `Event` object. You can call this function without stopping the StreetPass daemon process.

The data is stored as a `nn::cec::CecNotificationData` structure, so specify a buffer large enough to hold this structure in the *`pCecInfoBuffer`* and *`size`* parameters.

The `num` member of the `nn::cec::CecNotificationData` structure stores the number of information items in the structure. The `count` member stores the number of instances of StreetPass communication activated since the system was last booted. The `param` member stores information about the StreetPass data stored in the `nn::cec::CecNotificationParam` structure at position `num` from the start.

The `recvDate` member of the `nn::cec::CecNotificationParam` structure stores the date and time when the StreetPass data was received. The `cecTitleId` member stores the StreetPass ID, and the `messageId` member stores the StreetPass data's message ID.

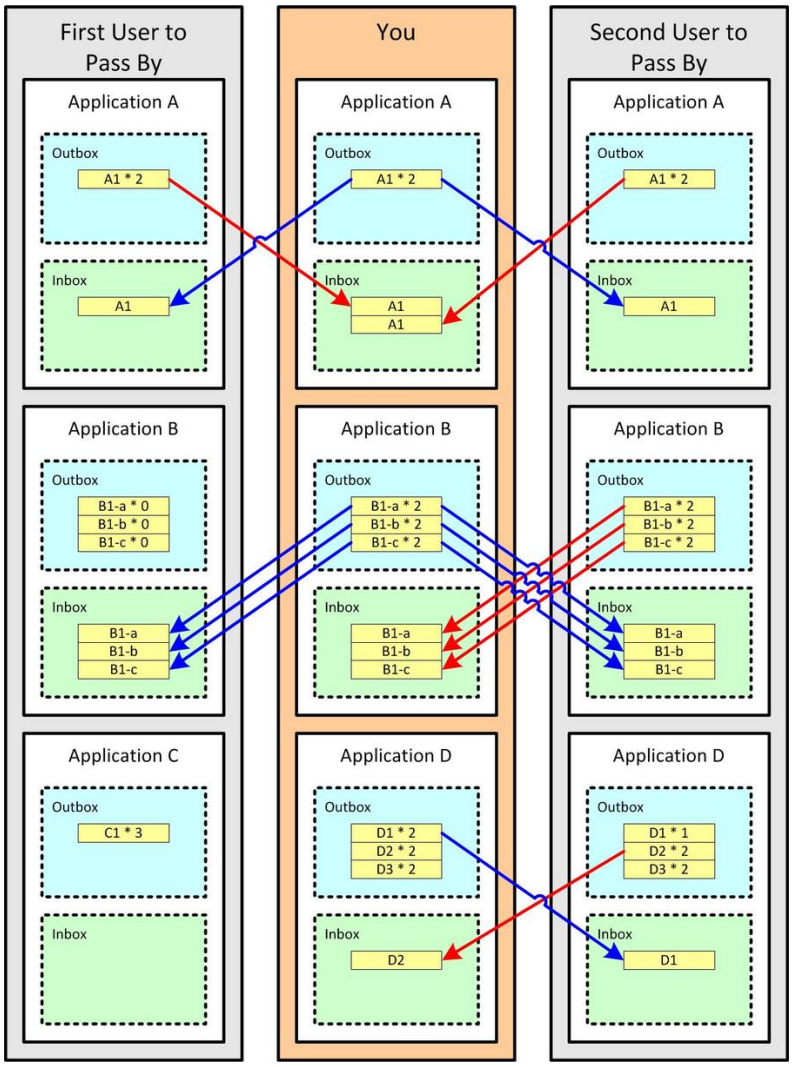
**Note:** The `messageId` member of the `CecNotificationParam` structure stores the message ID of the latest StreetPass data received in that StreetPass box. However, even when you receive the grouped StreetPass data, information for only one data item is stored.

You must open the StreetPass box or take some similar action to actually access the received data.

## 4.1.7. Example of Sending and Receiving StreetPass Data

The following figure shows an example of how data would be sent and received if you passed two other users and multiple applications had registered StreetPass data in StreetPass boxes.

**Figure 4-4. Example of Sending and Receiving StreetPass Data**




The blue lines show how data is sent, the red lines show how data is received, and the yellow boxes indicate the StreetPass data. The transmission mode for all of the StreetPass data is send/receive. Each StreetPass data item is given as a letter followed by: a number that represents a group, a hyphen and a letter that represents different data within the same group, and an asterisk

and a number that represents the transmission count. The transmission counts on your system show the values that were registered along with the data. The transmission counts on the other users' systems show the values at the moment they passed by you.

Application A had been registered by both of the users you encountered, so its data was sent and received twice.

Application B had been registered by both of the users you encountered. The first user had a transmission count of 0, however, so data was only sent from your system. Both you and the second user did not have a transmission count of 0, so data was both sent and received.

Application C had been registered only by the first user you encountered, so its data was neither sent nor received.

Data for application D was both sent and received with the second user you encountered, but you each received different data because your next item to send was D1 and the other user's next item to send was D2.

## 4.1.8. Efficient Settings

Because 3DS communication via StreetPass is handled as a background process, there may be StreetPass opportunities unrelated to the application currently running. However, the StreetPass settings include many items that are complicated and must be tweaked to efficiently send and receive StreetPass data. This section describes the settings related to sending and receiving data when an opportunity for StreetPass communication arises.

It also describes how to make effective use of the various settings. Features for a variety of purposes are provided by the library. To use these features, developers must constantly keep track of how often data is sent and received and between how many users.

### 4.1.8.1. Recommended Settings

Consider the normal case, where users select which data to exchange through StreetPass and register the data themselves. We recommend that you use the following settings so that it is easier for users to understand the results of StreetPass and to boost StreetPass transmission activity.

(1) Register only one data item.

You can avoid many of the restrictions inherent in the StreetPass specifications to make it more understandable to users.
If you want to register what seem to be many data items, first consolidate them into one item and register that item. Another way is to use grouping. If you do not use grouping and just register multiple items of data, only one data item ends up being sent and received in any one StreetPass.

(2) Set the transmission mode to **Send/Receive**.

When the transmission mode is set to **Send/Receive**, no opportunities will be lost because data is sent and received regardless of the state of either StreetPass inbox. **Exchange** mode has almost the same conditions for sending and receiving, but StreetPass is not established if there is no free space in either system's inbox or in some other cases unless both parties succeed in communication. In other words, the party with free space available will miss the opportunity to receive StreetPass data.
When the mode is **Send-only** or **Receive-only**, StreetPass will not occur even though two systems set to the same mode encounter one another. When users can select either of these modes, they are likely to tend to choose one mode over the other, resulting in missed opportunities for sending and receiving StreetPass data.

(3) Put no limit on number of times data can be sent.

If StreetPass is always operating, there are no lost opportunities. It is easier for users to understand the StreetPass settings when registered data remain effective until deleted. When a restriction is set, at some point the number of times that data can be sent will become 0, and the user may be unaware that distribution has ended. Because it is hard for a user to know when distribution has ended, other users will miss opportunities to receive StreetPass data when a user's transmission count has reached 0.

(4) Send to both friends and non-friends.

If there are no restrictions on who to send data to, no StreetPass opportunities are lost. While you might consider limiting sending to non-friends to increase opportunities with new people you happen to meet, data would no longer be exchanged with users with whom you have established friendships. Conversely, if you limit sending to only friends, your StreetPass can no longer be carried out with other regular users.

## 4.1.8.2. Registering Multiple Sets of Data

If you want to send multiple pieces of data to a single other user, we recommend not grouping them but rather packing them together into a single data item. By gathering the data this way instead of grouping, you can eliminate wasteful header information and reduce the overall size of the data.

When registering multiple pieces of data, you must note that only one item of data is sent at a time and that subsequent data items are not sent until the first has been sent. In other words, if the opportunity of StreetPass sending/receiving for the first data item is very low, subsequent data items will end up having to wait a long time for their turns.

The next StreetPass with the same other system will not occur until an average of 4 hours later, and the interval can be as long as 8 hours. For this reason, the second and subsequent data items will not reach that other system in series. If the user has registered multiple data items, the items that reach this other system depend on the items that were first in line on all occasions when the two systems made StreetPass encounters. Chance is a big factor, and the user may have trouble figuring out what's going on.

### 4.1.8.3. Using the Send-Target Setting

You can configure the send targets to limit who gets sent the StreetPass data. Combining this setting with grouping, you can send different data to friends and to non-friends.

To provide a specific example, set the data at the start of a group as data to be shared by all send targets, and then add two more data items (one to be sent to friends and the other to be sent to non-friends). StreetPass transmits that first data item first. Configure it in the way that minimizes lost opportunities.

### 4.1.8.4. Using the Propagation-Count Setting

One interesting aspect of StreetPass is that data registered by a distant user may be received after passing through multiple other users. This occurs when the propagation count for StreetPass data has been set to more than one. However, propagation ceases if data is deleted during its propagation, there is not enough available space in the outbox, or if the same data has already been received. There is also a possibility that data registered by the user may not be sent immediately depending on the send order due to registration of another user's data in the outbox.

You can use propagated data when you only want to disseminate special data inside a certain region, such as special deliveries from limited users such as operators or when applications register data at a low probability.

There is no special setting for data that is being distributed as StreetPass data registered by the user. If grouping is used, you can send it together with data registered by the user.

Note that you must register dummy data in a CTR system that is waiting for the arrival of propagating data if you are waiting to accept data being propagated without the user registering StreetPass data. For example, if an operator will be using this feature for special deliveries, set the operator side to send-only transmission mode, with a propagation count of 2 or more and a transmission count of 1, and set the side waiting for the propagating data to receive-only, with a propagation count of 1 and a transmission count of 1. This allows the registration of data being propagated to other users from operators and users encountered via StreetPass as an operator.

## 4.2. SpotPass

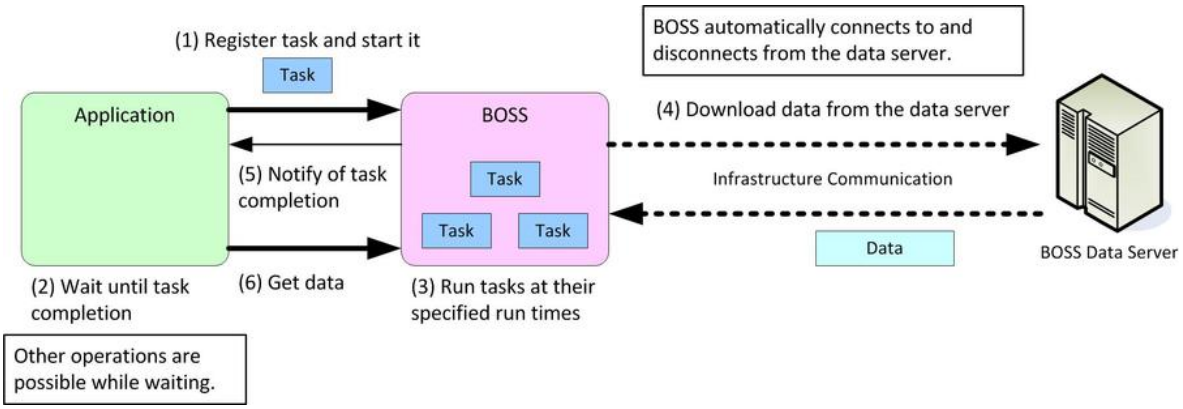
SpotPass is the name of a type of communication that uses the feature called BOSS (an acronym for "Background Online Service System"). Applications can use BOSS to handle background communication with Nintendo’s BOSS data servers.

An application uses BOSS by registering a task (which is a set of information describing the communication operations you want to perform) with BOSS and then starting the task. Tasks include

data such as execution interval and the execution count (number of remaining times to execute the task). BOSS then runs tasks regularly in the background based on this data.

> **Note:** The term *background* in this case refers to Sleep Mode and HOME Menu display. In other words, it refers to states where the application is not running.

**Figure 4-5. Downloading Data Using BOSS Features**




> **Note:** If the user has not accepted the EULA, tasks registered with BOSS will not run until the EULA has been accepted.

An application can find out whether a new data download task has finished by checking its task status, checking the task's execution result, or using notifications for newly arrived data events. This data can also be read via BOSS.
The following tasks are currently available.

- Nintendo Archive Download (NADL) tasks (4.2.2. Nintendo Archive Download (NADL) Tasks)
- Data upload tasks (4.2.4. Data Upload Tasks)
- Data Store upload tasks (4.2.5. Data Store Upload Tasks)
- Data Store download tasks (4.2.6. Data Store Download Tasks)

> **Note:** Data upload tasks and Data Store upload/download tasks require a server environment to be set up. Contact Nintendo (support@noa.com) before using this in a product.

Tasks are generally run in the background, but an application can prompt BOSS to run a registered task at any time. Note that this causes the task to run in the foreground.

In addition, a dedicated task registration function is provided to perform foreground execution of these tasks (for example, registering the tasks and executing immediately while the application is

running). This function registers tasks for immediate execution.

The following can be considered basic service examples of the use of BOSS.

- Regularly distributing notifications and extra data in the background (as NADL tasks)
- Distributing data that varies by company, using Nintendo Zone (as NADL tasks)
- Performing data exchanges between users in the background (as Data Store upload and download tasks)

## 4.2.1. Common Processes for All Tasks

This section describes common operations carried out for all task types.

Applications must carry out the following operations when using tasks.

- Initialize the BOSS library. (4.2.1.1. Initializing the BOSS Library)
- Prepare to register tasks. (4.2.1.2. Setting Task Properties and 4.2.1.3. Task Action Settings)
- Register and run tasks. (4.2.1.4. Registering and Running Tasks)
- Wait for tasks to complete. (4.2.1.5. Task Information and 4.2.1.6. Getting a Task List)
- Handle task results:
- Have the application handle the results (such as downloaded data).
- Change task registration information. (4.2.1.7. Changing a Task)
- Unregister tasks. (4.2.1.8. Unregistering a Task)
- Finalize the BOSS library. (4.2.1.9. Finalizing the BOSS Library)
- Handle BOSS library errors. (4.2.1.10. BOSS Library Error Handling)

In general, implement applications to follow these steps in the listed order.

### 4.2.1.1. Initializing the BOSS Library

You must first initialize the BOSS library by calling `nn::boss::Initialize` before using any BOSS features.

**Code 4-20. Initializing the BOSS Library**

```
nn::Result nn::boss::Initialize(void);
```

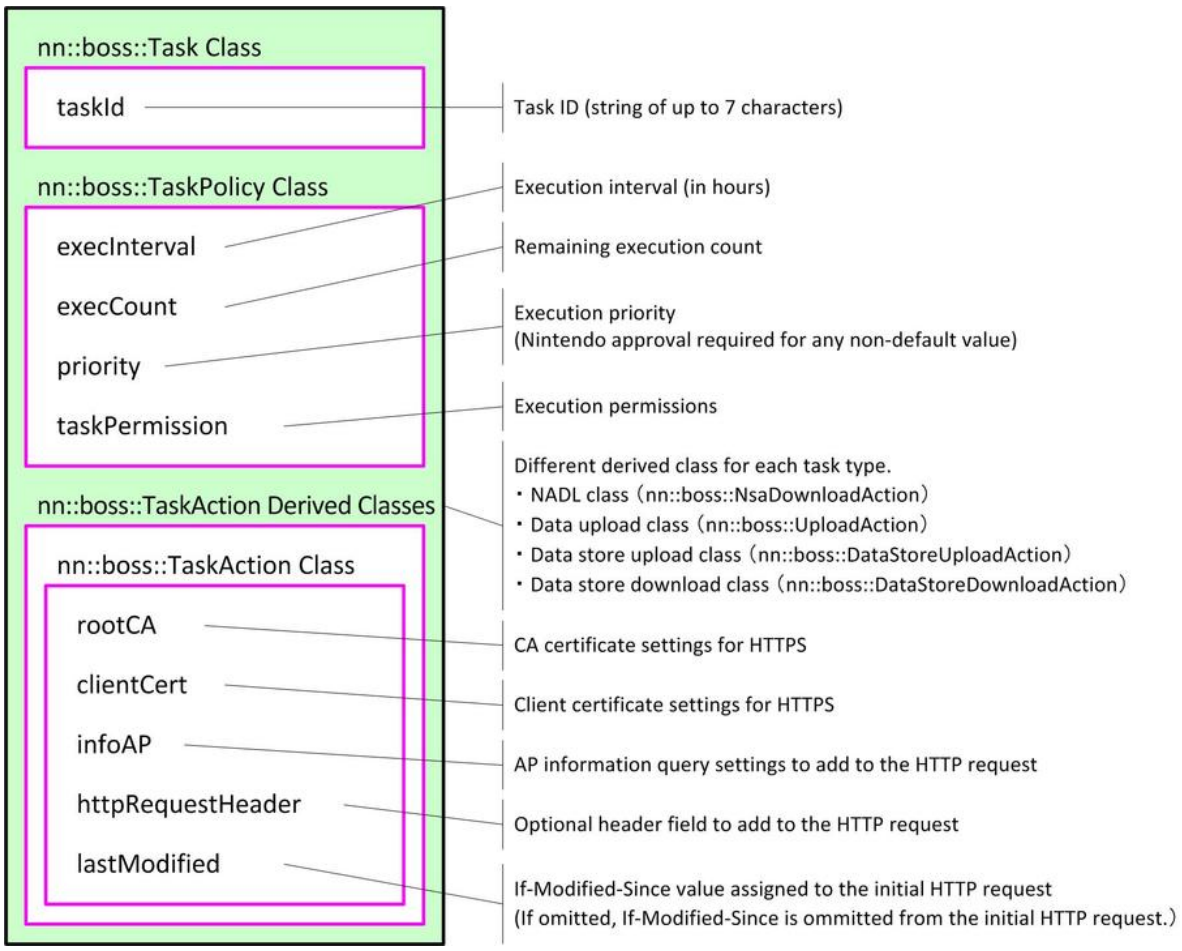
If you call the BOSS library functions before initializing the library, the functions return `nn::boss::ResultIpcNotSessionInitialized`.

## 4.2.1.2. Setting Task Properties

Set task properties using three classes. Then call the `nn::boss::RegisterTask()` function, passing in the three configured class instances as arguments, to register one task with the BOSS daemon.

- Basic information (`nn::boss::Task` class)
- Execution policy (`nn::boss::TaskPolicy` class)
- Actions (`nn::boss::TaskAction` class)

**Figure 4-6. Configuring Task Properties**




There are restrictions on the property values that may actually be used by the application.

The following table lists the restrictions that apply when registering NADL tasks.

**Table 4-8. Property Value Restrictions When Registering an NADL Task**

| Property | Restrictions on Valid Values |
|---|---|
| `taskId` | Use the task ID from your application to Nintendo. |
| `execInterval` | Use a value from 1 hour to 168 hours (one week).<br>To pass Lotcheck, your application must not use the function version provided for debugging purposes that allows you to set the execution interval in seconds. |

| | |
|---|---|
| execCount | Set an execution count from 1 to 100. |
| priority | Enter the default value (PRIORITY_MEDIUM).<br>Contact Nintendo (support@noa.com) if you need to set this to another value. |
| taskPermission | Enter the default value (TASK_PERMISSION_IN_PARENTAL_CONTROL).<br>When resetting properties using the SetProperty() function of the nn::boss::TaskPolicy class, always make sure to include TASK_PERMISSION_IN_PARENTAL_CONTROL. |
| infoAP | On normal tasks, do not set this to a value that provides access point information.<br>Individual data specific to particular Nintendo Zone locations and retail outlets can be distributed by providing the access point information. If you are using an NADL task that provides access point information, contact Nintendo (support@noa.com) in the early planning stages. |

### Basic Information (nn::boss::Task class)

The nn::boss::Task class for setting the basic information of a task includes a property for the task ID. Create an instance of the nn::boss::Task class and then call Initialize to set the task ID.

**Code 4-21. Initializing the nn::boss::Task Class**

```
class nn::boss::Task
{
    nn::Result Initialize(const char* pTaskId);
}
```

The task ID is a string of up to seven characters (not including the string terminator), and it can include uppercase and lowercase alphanumeric characters, the underscore, and the hyphen. This ID must be unique among all tasks belonging to the same application, so specify this value accordingly. Task IDs cannot be changed after they are assigned.

**Warning:** Be sure to apply to Nintendo (support@noa.com) for a task ID, and always use that approved task ID. If an unapproved task ID is set, the product will not pass Lotcheck.

### Execution Policy (nn::boss::TaskPolicy class)

The nn::boss::TaskPolicy class for setting a task's execution policy includes properties for the task execution interval and execution count. Create an instance of the nn::boss::TaskPolicy class and then call Initialize to set these properties. These values can be changed after initialization. For more information about changing the execution interval and execution count, see 4.2.1.7. Changing a Task.

**Code 4-22. Initializing the nn::boss::TaskPolicy Class**

```
class nn::boss::TaskPolicy
{
    nn::Result Initialize(u32 interval, u32 count);
}
```

Specify the execution interval for the *interval* parameter and the execution count for the *count* parameter. Interval values are in hours and can be from 1 to 168 (or 7 days). Count values can be from 1 to 100. Do not set UNLIMITED_COUNT as the count value. The count value is decremented every time the task is run. The task is no longer run after this value reaches 0.

**Note:** When the task is executed and the task finished running (the task state obtained with the nn::boss::Task::GetState() function is nn::boss::TASK_DONE or nn::boss::TASK_ERROR), the execution count is reduced by one. It is not decremented if the connection is interrupted during task execution.

**Warning:** The CTR-SDK also includes the debugging-only function InitializeWithSecInterval, which allows you to set the interval value in seconds. Note that any code setting this value in seconds will not pass Lotcheck.

Set and get all properties using the SetProperty() and GetProperty() member functions.

**Code 4-23. Setting and Getting Property Values**

```
class nn::boss::TaskPolicy
{
    nn::Result SetProperty(PropertyType type, void* pValue, size_t size);
    nn::Result GetProperty(PropertyType type, void* pValue, size_t size);
}
```

Specify the property to set or get in the *type* parameter. Specify a variable storing the property value for *pValue*, and specify its size in *size*.

**Table 4-9. Valid Property IDs for the nn::boss::TaskPolicy Class**

| Property ID | Type | Property |
|---|---|---|
| TASK_EXEC_INTERVAL | u32 | Execution interval (in seconds). |
| TASK_EXEC_COUNT | u32 | Execution count. |
| TASK_PERMISSION | TaskPermission | See the description following this table. |

The TASK_PERMISSION property identifier is accessed to restrict operations when Internet communication with other users ("Online Interaction") is restricted by Parental Controls. If a task has nothing set for this property, the task does not execute if Online Interaction is restricted by Parental Controls.

NADL tasks do not involve the exchange of data between users, so

TASK_PERMISSION_IN_PARENTAL_CONTROL is added to the TASK_PERMISSION identifier value by default. Consequently, leaving this value at the default ensures that NADL tasks will be processed even when Online Interaction is restricted by Parental Controls.

**Warning:** Any code in which the TASK_PERMISSION property identifier is not set to TASK_PERMISSION_IN_PARENTAL_CONTROL in a NADL task will not pass Lotcheck. When you change the TASK_PERMISSION identifier setting, you must include TASK_PERMISSION_IN_PARENTAL_CONTROL.

Although the TASK_PERMISSION property identifier can be used to set actions to perform when the EULA has not been accepted, make sure that tasks registered by the application do not perform actions when the EULA has not been accepted. If you register a task without setting actions, the task is not processed unless the EULA is accepted.

### Actions (nn::boss::TaskAction Class)

Use the relevant class that inherits nn::boss::TaskAction to set properties related to the operations to be carried out. The following tasks are currently available.

**Table 4-10. Task Types for the nn::boss::TaskAction Derived Classes**

| Derived Class | Task Type |
|---|---|
| nn::boss::NsaDownloadAction | Nintendo Archive Download (NADL) task |
| nn::boss::UploadAction | Upload task |
| nn::boss::DataStoreUploadAction | Data Store upload task |
| nn::boss::DataStoreDownloadAction | Data Store download task |

For more information about using tasks, and the valid property settings to use, see the individual task descriptions.
Upload tasks are provided on an individual basis. For this service, contact support@noa.com.

## 4.2.1.3. Task Action Settings

Settings can be performed for information provided when connecting to a server and certificates using the nn::boss::TaskAction class, which perform settings specific to task actions.

**Code 4-24. Functions for Setting Task Actions**

```
class nn::boss::TaskAction
{
```

```
        nn::Result SetApInfo(ApInfoType info);
        nn::Result SetCfgInfo(CfgInfoType info);
        nn::Result AddHeaderField(const char* pLabel, const char* pValue);
        nn::Result SetLastModifiedTime(const char* pLastModifiedTime);
        nn::Result SetRootCa(u32 inCaCertId);
        nn::Result SetClientCert(u32 inClientCertId);
    }
```

All properties of the `nn::boss::TaskAction` class can be set and obtained using the `SetProperty()` and `GetProperty()` functions. These functions can be defined in the same way as the `nn::boss::TaskPolicy` class definition.

## Providing Access Point Information

Call the `SetApInfo()` function to specify information to be provided to an access point (AP) for an HTTP query. This allows you to select AP information to be sent from the Nintendo Zone access point to the BOSS data server, when infrastructure communication is occurring via a Nintendo Zone access point using AP information provided in the form of an HTTP request query. The query which is provided can be used only when you need to add changes to the task processing when connecting via the Nintendo Zone. One example would be downloading different data only when connecting to the Nintendo Zone.

**Warning:** Items with access point information provided do not pass through Lotcheck. If you are using tasks with access point information provided, make sure to contact support@noa.com early in the product planning process.

**Table 4-11. Access Point Information Provided to an HTTP Request**

| Query | Information Provided |
|---|---|
| `ap` | Information that can be identified at the access point level. |
| `apgroup` | Information that can be identified at the access point group level (such as company name). |

Specify access point information to provide from the following definitions (select only one).

**Table 4-12. Specifying AP Information to Provide to an HTTP Request**

| Definition | Description |
|---|---|
| `APINFOTYPE_AP` | Provide ap.<br>Specify if applying to distribute according to Nintendo Zone operators. |
| `APINFOTYPE_APGROUP` | Do not specify anything for these arguments in normal circumstances. |

## Providing System Settings Information

Call the `SetCfgInfo()` function to specify the provision of System Settings information to an HTTP query. This allows you to provide system information to a query from an HTTP request. This may be used (for example) for downloading different data using the language setting of a system connected to the client server.

> **Note:** This feature is specific to the BOSS service, using a client server. Providing System Settings information with a BOSS service, using a Nintendo server, has no effect.
>
> If you are considering an independent server using the provided information, consult with Nintendo (support@noa.com).

The following definitions are used in specifying system information to be provided. Multiple information items can be granted simultaneously, by setting multiple values using a bitwise OR.

**Table 4-13. Specifying System Settings Information to Provide to an HTTP Request**

| Definition | Description |
|---|---|
| `CFGINFOTYPE_NONE` | Do not provide System Settings information. |
| `CFGINFOTYPE_COUNTRY` | Provide country code in System Settings information. |
| `CFGINFOTYPE_LANGUAGE` | Provide the language code in System Settings information. |

### Adding an HTTP Request Header

You can include a proprietary HTTP request header in the connection, by calling the `AddHeaderField()` function. To add an HTTP request header, its label must have a maximum length of 32 characters and its value a maximum length of 256 characters. The maximum number of headers allowable is 3.

### Specifying Final Update Time at the First Execution

Call the `SetLastModifiedTime()` function to specify the string value, to enter in the **If-Modified-Since** header field for the first time the task is executed. When this specification is not performed, an **If-Modified-Since** header field is not provided the first time the task is executed. For the second and subsequent times the task is executed, an **If-Modified-Since** header field is provided containing the time of the last update for data obtained during the prior execution (the value in the HTTP response's **Last-Modified** field). When executing, if the task does not result in acquiring the data due to its not having been updated (for example, if a "304 Not Modified" status is returned by the server), the execution count is reduced by 1.

In addition, properties setting functions specific to device-internal certificates may also be used. (These functions are inherited from the `nn::boss::TaskAction` class.)

**Code 4-25. Functions for Setting Properties Specific to Device-Internal Certificates**

```
class nn::boss::TaskAction
{
    nn::Result SetRootCa(u32 inCaCertId);
    nn::Result SetClientCert(u32 inClientCertId);
}
```

You can configure the settings that determine which certificates embedded in a device are used by using the `SetRootCa()` and `SetClientCert()` functions. Up to three root CA certificates can be set, whereas only one client certificate can be set.

### Setting Proprietary Certificates

The following functions can be used to set a proprietary certificate when using a proprietary root CA certificate or client certificate to connect when executing a task. There is no need to call them when not using proprietary certificates.

**Code 4-26. Setting Proprietary Certificates**

```
nn::Result nn::boss::RegisterPrivateRootCa(
                        const u8* pCertData, size_t certDataSize);
nn::Result nn::boss::RegisterPrivateClientCert(
                        const u8* pCertData, size_t certDataSize,
                        const u8* pPrivateKeyData, size_t privateKeyDataSize);
```

A proprietary certificate and a certificate embedded in a device can be used in combination. Only one certificate (either the root CA certificate or the client certificate) can be set as the proprietary certificate. When duplicate settings occur, the last certificate set is enabled.

To use a proprietary certificate, call the `SetPrivateRootCa()` or `SetPrivateClientCert()` member function from the `TaskActionBase` class. Only the root CA certificate can be used in combination with a certificate embedded in a device.

**Note:** Registering tasks to connect to servers other than the BOSS data server provided by Nintendo is currently not permitted.

## 4.2.1.4. Registering and Running Tasks

Call `nn::boss::RegisterTask` with each of the four task property classes as arguments to register a task.

**Code 4-27. Registering a Task**

```
nn::Result nn::boss::RegisterTask(nn::boss::Task* pTask,
                                  nn::boss::TaskPolicy* pPolicy,
                                  nn::boss::TaskAction* pAction,
                                  nn::boss::TaskOption* pOption=NULL,
```

```
                                    u8 taskStep=DEFAULT_STEP_ID);
```

**Note:** The *pOption* and *taskStep* parameters are for future expansion and are currently unsupported. Do not specify anything for these arguments.

Just registering a task will not cause it to be executed. You must also call the `nn::boss::Task` class's `Start` or `StartImmediate` member functions to tell the BOSS daemon to run the task.

**Code 4-28. Running, Canceling, and Waiting to Finish a Task**

```
class nn::boss::Task
{
    nn::Result Start(void);
    nn::Result StartBgImmediate(void);
    nn::Result StartImmediate(void);
    nn::Result Cancel(void);
    nn::Result WaitFinish(void);
    nn::Result WaitFinish(const nn::fnd::TimeSpan& timeout);
}
```

Tasks marked for execution with the `Start()` function are scheduled and run by the BOSS daemon in the background. The daemon automatically handles Internet connections for tasks run in the background. If a connection is lost while a background task is running, the daemon pauses the task. Tasks in a resume state are resumed from the point of interruption the next time an Internet connection can be made.

`Tasks begun with the Start()` function can be halted or removed from the scheduling queue by calling the `Cancel()` function. Call `Start` again to tell the daemon to add a halted task back into the scheduling queue.

Call `StartImmediate` to tell the daemon to run a task in the foreground. Tasks marked for execution with the `StartImmediate()` function are run right away. Because such tasks are run in the foreground, the application must handle connecting to the Internet, as the BOSS daemon does not handle these connections. Unlike running with the `Start()` function, tasks run using `StartImmediate` return an error when the connection is lost while the task is running. Interrupted tasks cannot be resumed.

You can also use `StartImmediate` to run a task already queued up using the `Start()` function, without having to wait until the execution interval has passed. However, note that this again runs the task in the foreground, and that the application must handle connecting to the Internet.

**Note:** While a task is executing, other tasks will not execute (even if the start-execution date of another task has been reached). However, if you execute a task immediately using `StartImmediate()`, tasks executing in the background are stopped and resume after the immediate-execution task is finished.

**Warning:** When a task originally started with an instruction from the `Start()` function is

executed using the `StartImmediate()` function, the task transitions to the retry state if its original execution is still in progress. This does not cause a problem as BOSS normally does not execute tasks while the application is running. However, when BOSS has been authorized to run while the application is running (this can be set in the NDM library), developers must take this into consideration. Handle applications of this sort as follows.

- Cancel the task by calling the `Cancel()` function, and then execute the `StartImmediate()` function.

Call the `StartBgImmediate()` function to tell the daemon to start tasks in the background immediately. Use this function to **begin a task's first scheduled execution immediately**. If you call the `StartBgImmediate()` function on a task that was previously scheduled to run as the result of a call to the `Start` or `StartImmediate()` functions, the task is not run immediately a second time. If you want the task to run immediately every time it is called, use the `StartImmediate()` function instead. If you need to provide additional background instructions for immediate execution to a task that has already been scheduled, cancel its execution using the `Cancel()` function, and then call the `StartBgImmediate()` function. Note that calling `StartImmediate()` cancels the execution of other tasks, whereas `StartBgImmediate()` runs the targeted task after any currently running tasks complete.

Call the `WaitFinish()` function to wait for one execution of a task to complete. Use this for the application to wait for task completion in cases, such as: tasks with an execution count of one, or tasks run immediately with `StartImmediate`. Calling the `StartImmediate()` function cancels execution of other tasks, while the `StartBgImmediate()` function executes the task after other tasks have completed execution. We recommend calling `WaitFinish` using either a dedicated thread or using the overloaded version and specifying a value for the *`timeout`* parameter. When specifying a timeout value, the function returns with `ResultWaitFinishTimeout` if the task has not completed within the timeout period, passing control back to the application.

The arrival of new data can be used as a method of waiting for task execution to complete. Use `nn::boss::GetNewArrivalFlag` or `nn::boss::RegisterNewArrivalEvent` to check newly arrived data. For more information about these functions, see 4.2.2.6. Checking the New Arrival Flag and Waiting for a Data New Arrival Event.

There are also methods of polling task status using `nn::boss::GetState()` and `nn::boss::GetStateDetail()`.

### Registering Immediate Execution Tasks

The `nn::boss::RegisterImmediateTask()` function is provided for registering tasks that can only be executed immediately. **Always register immediate execution tasks using this function, and execute them in the foreground.The guidelines restrict the number of tasks that can be registered in one application, but immediate execution tasks registered with this function are not counted among those tasks.**

**Code 4-29. Registering Immediate Execution Tasks**

```
nn::Result nn::boss::RegisterImmediateTask(
        nn::boss::Task* pTask, nn::boss::TaskAction* pAction,
        nn::boss::TaskPolicy* pPolicy=NULL, nn::boss::TaskOption* pOption=NULL,
        u8 taskStep=DEFAULT_STEP_ID);
```

For *pTask* instead of using the `nn::boss:Task` class, specify its subclass, the `nn::boss::FgOnlyTask` class. When this class is used, task IDs registered for immediate execution defined in `nn::boss::FG_ONLY_TASK_ID` are automatically specified. It is not necessary to specify the task ID. Even if execution count and execution intervals have been set in *pPolicy*, these settings are disabled.

Task execution must occur with the `nn::boss::FgOnlyTask::StartImmediate()` function. Executing a task with any other function returns an error.

### Maximum Number of Tasks That Can Be Registered and Automatic Unregistering

The maximum number of tasks that can be registered in BOSS is 127. (Tasks for immediate execution are not included.) This number represents the maximum value of the total number of tasks registered by all applications (including embedded applications). The maximum number of tasks that can be registered by a single application is limited by the *CTR Guidelines*. If an application attempts to register a new task when the number of registered tasks has already reached the maximum, BOSS automatically unregisters a task for which the execution count is 0 and accepts the registration of the new task. This is not just limited to tasks of the application attempting registration. A search is made among all tasks for a task to unregister. If there is no task with an execution count of 0, the product of the execution count and the execution interval is calculated, the task with the smallest resulting value is unregistered, and registration of the new task is accepted.

If an application has not been run for a long time, its previously registered tasks might get unregistered by BOSS using this automatic unregistering process. For this reason, applications that use BOSS must be able to handle cases where their tasks have been unregistered. You must implement a process that (for example) checks whether an application has any registered tasks when that application starts up and, if tasks are not registered, prompts for user confirmation to register the tasks.

## 4.2.1.5. Task Information

Use the member functions of the `nn::boss::Task` class to get information about a task, such as its state or execution result.

**Code 4-30. Getting Task Information**

```
class nn::boss::Task
{
    TaskServiceStatus GetServiceStatus(void);
```

```
    TaskStateCode     GetState(bool acknowledge=false,
                               u32* pCount=NULL, u8* pStepID=NULL);
    u32               GetHttpStatusCode(u32* pCount=NULL, u8* pStepID=NULL);
    TaskResultCode    GetResult(u32* pCount=NULL, u8* pStepID=NULL);
    nn::Result        GetStateDetail(TaskStatus* pStatus,
                                     bool acknowledge=false, u8* pStepID=NULL,
                                     u8 taskStep=CURRENT_STEP_ID);
    nn::Result        GetError(TaskError* pTaskError, u8* pStepID=NULL,
                               u8 taskStep=CURRENT_STEP_ID);
    nn::Result        GetInfo(TaskPolicy* pPolicy, TaskAction* pAction,
                              TaskOption* pOption, u8
taskStep=CURRENT_STEP_ID);
    nn::Result        GetActivePriority(TaskPriority* pPriority);
}
```

For the functions that have a *pCount* parameter, pass a pointer to a u32 variable instead of NULL to have the function return the execution count for a task. The *pStepID* parameter cannot be used in the current version.

> **Warning:** The task execution count and information for tasks acquired returned in *pCount* can be individually obtained in the library. There is a slight time lag as a result, so information does not necessarily match precisely.

Call the GetServiceStatus() function to get the state of the server the task is connecting to (such as SERVICE_AVAILABLE or SERVICE_TERMINATED). The server sends its state when the task runs.

A return value of SERVICE_AVAILABLE indicates that the service is running. A return value of SERVICE_TERMINATED indicates that the service is running. The function returns a value of SERVICE_UNKNOWN when a task has not yet been executed or when a connection attempt fails out in a network error. If the function itself fails, it returns a value of GET_SERVICE_STATUS_ERROR.

> **Warning:** The system checks whether a service has terminated by checking the response from the server when the task runs to determine whether the response includes a flag indicating that the service has terminated (the service termination flag). This flag is set by the application developer using the BOSS data server.

Call the GetState() function to get the state of the task. If the function itself fails, it returns a value of GET_TASK_STATE_ERROR. To ensure that accurate execution results for a task can be obtained, task execution results are stored from the time a task completes execution until the next task executes (in other words, while the status is TASK_WAITING_TIMER). The GetState() function returns the execution result of the last task (TASK_DONE, TASK_ERROR, or TASK_RETRY) even though the actual current status is TASK_WAITING_TIMER.

To get the actual current status (rather than the result being stored) call the GetState() function while specifying true in the *acknowledge* parameter. Note that the stored result is released and the actual status (TASK_WAITING_TIMER) can be obtained when the task result is obtained using the GetState() function afterward.

**Table 4-14. Task States**

| Definition | Description |
|---|---|
| TASK_STOPPED | Stopped, and excluded from scheduling. |
| TASK_WAITING_TIMER | Waiting for the task execution interval to pass. |
| TASK_WAITING | The task execution interval has passed, but the task is still waiting to run for some reason (such as not being connected to the Internet or because another higher-priority task is running). |
| TASK_RUNNING | The task is now running. |
| TASK_PAUSED | Task execution is paused. |
| TASK_REGISTERED | The state right after registering a task, but before starting it. |
| TASK_DONE | The task has finished. |
| TASK_ERROR | An unrecoverable error occurred while the task was running. |
| TASK_RETRY | Connection was lost during execution and the task is now waiting to retry (resume). |

Call the GetHTTPStatusCode() function to get the HTTP status code when a task is running. If the function itself fails, it returns a value of U32_CANNOT_GET_DATA.

Call the GetResult() function to get the result of a task. If the function itself fails, it returns a value of GET_TASK_RESULT_ERROR. For the return value definitions, see the definition of the TaskResultCode enumerated type in the nn/boss/boss_Const.h file. The return value called during the BOSS library and returned as an error by functions from other libraries becomes a value larger than the value that exists in the TaskResultCode enumerated type UNKNOWN_ERROR + the module ID for the module that generated the error. For the module ID definitions, see nn/Result.h. FS_UNKNOWN_ERROR is returned when an error occurs for file operations at task execution due to any of the following reasons.

- There is no space available on the SD card.
- The SD card is write-protected.
- An SD card is not inserted.
- A different SD card than the one used at the time the task was registered is inserted, meaning that BOSS storage for writing and reading task data is not available. (BOSS storage is covered in the sections that follow.)
- The SD card is damaged.

Call the GetStateDetail() function to get a task's state. The difference between this function and the GetState() function is that you can use the GetProperty() function as an instance of the nn::boss::TaskStatus class returned in the *pStatus* parameter to get more detailed state information.

**Table 4-15. Property IDs Specifiable for the nn::boss::TaskStatus Class**

| Property ID | Type | Property |
|---|---|---|
| | | |

| | | |
|---|---|---|
| TASK_STATE_CODE | TaskStateCode | Task state (same as for GetState). |
| TASK_RESULT_CODE | TaskResultCode | Execution result (same as for GetResult). |
| TASK_COMM_ERROR_CODE | u32 | Communication error code (same as for GetHttpStatusCode). |
| TASK_CURRENT_PRIORITY | u32 | Current task execution priority. |
| TASK_EXECUTE_COUNT | u32 | Current task execution count.<br>(It takes some time for this to reflect the most recent value, so we recommend getting the property of the same name from the nn::boss::TaskPolicy class instead.) |
| TASK_PENDING_TIME | u32 | Time in the task execution interval (in seconds). |
| TASK_START_TIME | s64 | Task start time (in seconds, starting from 2000-01-01). |
| TASK_PROGRESS | u32 | Size of downloaded data. |
| TASK_DATA_SIZE | u32 | Total size of the data to download (value of the Content–Length HTTP response field). |
| TASK_SERVICE_STATUS | TaskServiceStatus | Service availability status. |
| TASK_SERVICE_TERMINATED | bool | Service-terminated flag. |
| TASK_LAST_MODIFIED_TIME | char[MAX_LASTMODIFIED_LENGTH] | Last time data to download was modified (value of the Last–Modified HTTP response field). |

Call the GetError() function to get any error that occurred during task execution. The difference between the GetCommErrorCode() and GetResult() functions is that you can use the GetProperty() function as an instance of the nn::boss::TaskError class, returned in the *pTaskError* parameter, to get more detailed information.

**Table 4-16. Property IDs Specifiable for the nn::boss::TaskError Class**

| Property ID | Type | Property |
|---|---|---|
| TASK_ERROR_RESULT_CODE | TaskResultCode | Task result (only for errors; same as for GetResult). |
| TASK_ERROR_CODE | bit32 | Cause of an error in task execution. |
| TASK_ERROR_MESSAGE | char[MAX_ERROR_MESSAGE] | Communication error message during task execution. |

Call the `GetActivePriority()` function to get a task's current execution priority.

## 4.2.1.6. Getting a Task List

Call the `nn::boss::GetTaskIdList()` function to get a list of tasks registered by the application.

**Code 4-31. Getting a Task List**

```
nn::Result nn::boss::GetTaskIdList(nn::boss::TaskIdList* pTaskIdList);
```

For the *`pTaskIdList`* parameter, specify a pointer to an instance of the `nn::boss:TaskIdList` class (generated by the application). An instance is roughly 1 KB in size, so take note of the stack and heap sizes when generating an instance. The `nn::boss::TaskIdList` class is defined as follows.

**Code 4-32. nn::boss::TaskIdList Class**

```
class nn::boss::TaskIdList
{
    explicit TaskIdList(void);
    virtual ~TaskIdList(void);
    u16 GetSize(void);
    char* GetTaskId(u16 index);
}
```

Call the `GetSize()` function to get the number of tasks stored in a list. Call the `GetTaskId()` function and specify an index to get the ID of a task stored in a list. The function returns NULL if the specified index is out of bounds.

**Note:** The `nn::boss::GetStepIdList()` function is not currently usable.

## 4.2.1.7. Changing a Task

Call the `nn::boss::ReconfigureTask()` function to change a task's property settings. You can only change those properties that can be set by the `nn::boss::TaskPolicy()` function. Do not specify a step ID.

**Code 4-33. Changing a Task**

```
nn::Result nn::boss::ReconfigureTask(nn::boss::Task* pTask,
                                     nn::boss::TaskPolicy* pPolicy,
                                     u8 taskStep=DEFAULT_STEP_ID);
```

Use the `nn::boss::Task` member functions to change a task's execution count or execution interval.

**Code 4-34. Setting Properties Using the nn::boss::Task Class**

```
class nn::boss::Task
{
    nn::Result UpdateInterval(u32 interval);
    nn::Result UpdateIntervalWithSec(u32 intervalSec);
    nn::Result UpdateCount(u32 count);
    u32 GetInterval(void);
    u32 GetIntervalSec(void);
    u32 GetCount(void);
}
```

You can call the `UpdateIntervalWithSec()` function to specify an execution interval in seconds, but note that this function is only for debugging purposes.

## 4.2.1.8. Unregistering a Task

Call the `nn::boss::UnregisterTask()` function to unregister a task.

**Code 4-35. Unregistering a Task**

```
nn::Result nn::boss::UnregisterTask(nn::boss::Task* pTask,
                                    u8 taskStep=DEFAULT_STEP_ID);
```

When you unregister a task, the working files created for that task in BOSS storage are deleted. The function returns an error (`nn::boss::ResultFileAccess`) if it fails to delete the working files (for example, because the SD card was removed), but the task itself will be successfully unregistered. You are not required to handle this error because the working files will still be deleted when the expanded save data is deleted. Although you cannot delete working files directly, you can delete them by reregistering a task with the same task ID and then unregistering it.

> **Note:** The *taskStep* parameter cannot be used at present. Do not specify anything for this parameter.

## 4.2.1.9. Finalizing the BOSS Library

After you finish using BOSS, you must call the `nn::boss::Finalize()` function to finalize the BOSS library.

**Code 4-36. Finalizing the BOSS Library**

```
nn::Result nn::boss::Finalize(void);
```

## 4.2.1.10. Handling Errors in the BOSS Library

There are four broad ways to handle BOSS library errors.

- Errors due to programming.
  These are assertion-level errors caused by how the BOSS library is being used. For these kinds of errors you must revise your application.
  `nn::boss::ResultInvalidTaskId`, `nn::boss::ResultIpcNotSessionInitialized`, and `nn::boss::ResultStorageNotFound` are these kinds of errors.
  These are just some examples. For information about the other results that correspond to this situation, see the API reference.
- Errors that the various API functions must handle on their own.
  `nn::boss::ResultNsDataListSizeShortage`, `nn::boss::ResultNsDataListUpdated`, and `nn::boss::ResultTaskIdAlreadyExist` are these kinds of errors.
  These are just some examples. For more information about the other results that correspond to this situation, and the actions to take, see the API reference.
- File access error.
  The BOSS library functions can return this kind of error due to file system errors.
  `nn::boss::ResultFileAccess` is the only example.
  Specify `nn::boss::FS_UNKNOWN_ERROR` for `nn::boss::TaskResultCode`, get the error code using the `nn::boss::GetErrorCode()` function, and show the error using the error/EULA applet.
- Other errors
  These are unexpected errors.
  Specify `nn::boss::SEVERE_ERROR` for `nn::boss::TaskResultCode`, get the error code using the `nn::boss::GetErrorCode()` function, and show the error using the error/EULA applet.

Following is an example of error handling with the `nn::boss::GetNsDataIdList()` function. Every API function differs in the errors due to programming and the errors that the function must handle on its own, so make sure that you check the API function references.

**Code 4-37. Error Handling With the nn::boss::GetNsDataIdList Function**

```
static const u32 MAX_DATA_ID = 32;
u32 idBuf[MAX_DATA_ID];// Buffer for storing the serial ID. Prepared by the
application.A large buffer can store many serial IDs en bloc.
nn::boss::NsDataIdList serialIdList(idBuf, MAX_DATA_ID);

do
```

```
{
    nn::Result result = nn::boss::GetNsDataIdList(nn::boss::DATA_TYPE_ALL,
&serialIdList);
    if(result.IsFailure())
    {
        if(result == nn::boss::ResultStorageNotFound() ||
                result == nn::boss::ResultInvalidNsDataIdList() ||
                result == nn::boss::ResultIpcNotSessionInitialized())
        {
            // Errors due to programming. (Differs for every API function, so
see the API references for more information.)

            // Errors due to implementation mistakes in the application, like
BOSS storage is not registered, the pointer to the NS data list information is
NULL, or the BOSS library is not initialized.
            // These are assertion-level errors, so make sure that you fix them
at the development stage.
        }
        else if(result == nn::boss::ResultNsDataListSizeShortage())
        {
            // Errors that the various API functions must handle on their own.
(Differs for every API function, so see the API references for more
information.)

            // Cases where the list of IDs was retrieved successfully, but not
all IDs could be stored in NsDataIdList.
            // If necessary, run nn::boss::GetNsDataIdList again.
            // In this example, GetNsDataIdList is called again without leaving
do-while.
        }
        else if(result == nn::boss::ResultNsDataListUpdated())
        {
            // Errors that the various API functions must handle on their own.
(Differs for every API function, so see the API references for more
information.)

            // Cases where NS data was added or deleted from BOSS storage while
the list of IDs was being retrieved.
            // Initialize the list, and get the list of serial IDs again from
the start.
            serialIdList.Initialize();
            result = nn::boss::ResultNsDataListSizeShortage();// Updates result
values without leaving the do-while loop.
            continue;
        }
        else if(result == nn::boss::ResultFileAccess())
        {
            // Handling a file access error.

            // Use the error/EULA applet to show the error code indicating a
file access error (nn::boss::FS_UNKNOWN_ERROR).
            u32 errCode = 0;
            nn::Result getErrCodeResult = nn::boss::GetErrorCode( &errCode,
nn::boss::FS_UNKNOWN_ERROR);
            NN_UTIL_PANIC_IF_FAILED(getErrCodeResult);// Fails only if the
boss::GetErrorCode function is used incorrectly. For more information, see the
API reference.

            // Error/EULA applet codes have been omitted.

            break;
        }
        else
        {
            // Handling an unexpected error.
            // Use the error/EULA applet to show the error code indicating
another error (nn::boss::SEVERE_ERROR ).
            u32 errCode = 0;
            nn::Result getErrCodeResult = nn::boss::GetErrorCode( &errCode,
nn::boss::SEVERE_ERROR);
            NN_UTIL_PANIC_IF_FAILED(getErrCodeResult);// Fails only if the
```

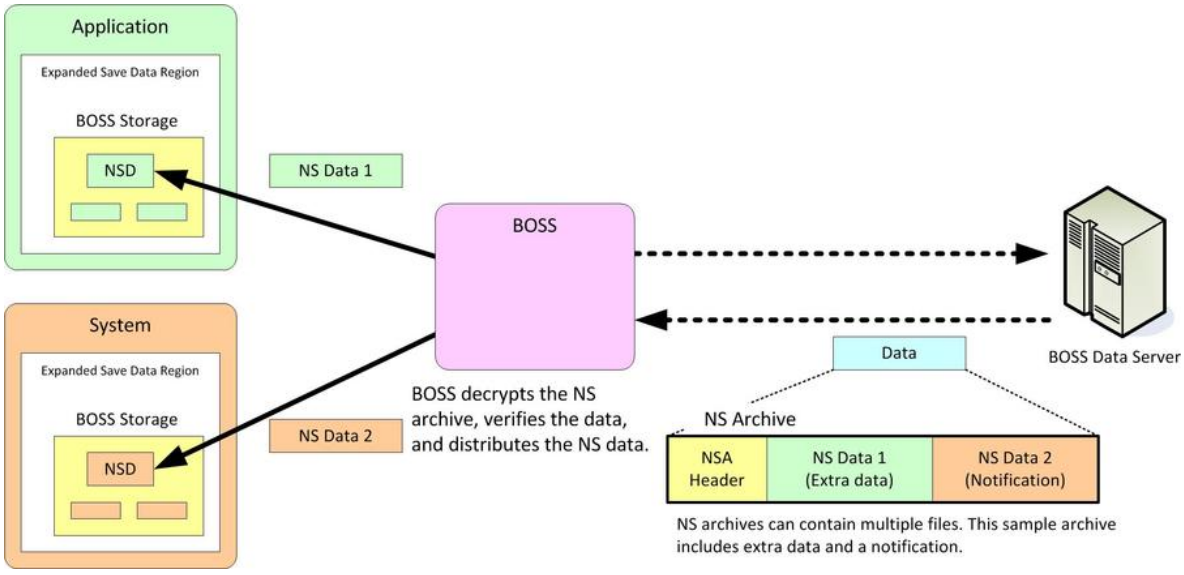
```
boss::GetErrorCode function is used incorrectly. For more information, see the
API reference.

            // Error/EULA applet codes have been omitted.

            break;
        }
    }

    {
        // Process to perform on the obtained serialIdList.
    }
}while(result == nn::boss::ResultNsDataListSizeShortage());// In this example,
GetNsDataIdList is executed when ResultNsDataListSizeShortage.
```

## 4.2.2. Nintendo Archive Download (NADL) Tasks

An NADL task is a BOSS task that downloads data from the BOSS data server via HTTP or HTTPS.

Data must be registered on the BOSS data server for an application to download it. Data registered on the BOSS data server is converted to the proprietary Nintendo Serendipitous (NS) archive format. NS archives can contain multiple data files. Accordingly, you can register multiple data files in a single archive on the BOSS server, so that users can download multiple files (such as application data files and related notifications) by running a single task.

**Figure 4-7. Process Flow When Downloading Data Using an NADL Task**




For information about using the BOSS server, log in to the BOSS data server, and open Help. Help also includes information about distributed data and operations, so read it before operating the BOSS Data Server.

The management tool performs the following primary operations.

- Creation of notifications and distributed data
- Management of the distribution schedule for notifications and distributed data

- Test distributions during application development
- Obtaining the distribution log that is sent to users after the actual start

To use the BOSS Data Server, apply to be a "User who Can Access the BOSS Data Server" when applying to use BOSS tasks on the Online title MAnagement System (OMAS). Upon approval of your application, access information for the BOSS Data Server is sent in an email.

## 4.2.2.1. NS Archives and NS Data

All data included in an NS archive (NS data) is encrypted, signed, and hashed. As a result, BOSS can handle not only NS data decryption, but also data verification to check for tampering or spoofing (to ensure that no invalid data is downloaded).

You can set a new arrival flag for an NS archive. After this flag is set, applications display an update indicator on their icons after downloading the archive. The new arrival flag is set separately for each application managed using BOSS, and the system checks for the new arrival flag and displays the update indicator accordingly. The system handles all of this, so applications do not need to do anything about this update indicator display.

When the system downloads NS data that includes notifications, it displays the update indicator on the **Notifications** applet (even if the new arrival flag is OFF, disabled). However, this only applies to the update indicator on the **Notifications** applet; the update indicator behavior for the application's icon is still controlled by the new arrival flag.

NS data can include the following.

**Table 4-17. NS Data Types**

| Type | Description |
|---|---|
| Extra data. | Data for various extras used in an application, such as extra items or game levels. |
| Notifications | Notifications sent via BOSS. |
| Downloaded contextual banner data. | Data for replacing the text or textures displayed in CTR title banners. |
| Data Store data | Data downloaded with the Data Store download task. |

Each NS data file includes a 32-bit data type designator. The data type consists of two types. The most-significant 16 bits specify the global data type indicating an NS data file, and the least-significant 16 bits specify the private data type. Global data types are shared among all applications and must be specified from the table above. Private data type values can be freely assigned by each application. This private data type is used as a search term when searching NS data, as described in 4.2.2.6. Checking the New Arrival Flag and Waiting for a Data New Arrival Event. Properly planned private data types allow you to search NS data more efficiently.

Note that NS data is discarded if there is no BOSS storage for the receiving application when the BOSS library distributes the NS data to its various storage locations. For example, this situation

applies when the application has never been started, the application has been started before but has never registered BOSS storage, or when the user has deleted the application's extra data. NS data also includes the unique ID of the target application embedded within it. In the NS data's unique ID, the unique ID of the application using BOSS is embedded based on the server. For that reason, developers cannot specify an arbitrary unique ID (target) to data. Downloaded NS data is stored in the BOSS storage of the application whose unique ID is embedded within, regardless of which application downloaded the task that was used to download.

**Warning:** Make sure that you avoid specifying the wrong URL upon task registration or downloading NS data targeting other applications.

Each NS data file is also assigned a serial ID, which must be unique among all NS data for a specific application, and a version number. Both are 32-bit values. Serial IDs are used by applications to specify NS data to read or delete. Versions are used by BOSS when deciding whether to update NS data. When BOSS downloads data with the same serial ID as existing NS data, BOSS then compares version numbers and only updates (that is, overwrites) the NS data if the downloaded data's version number is newer.

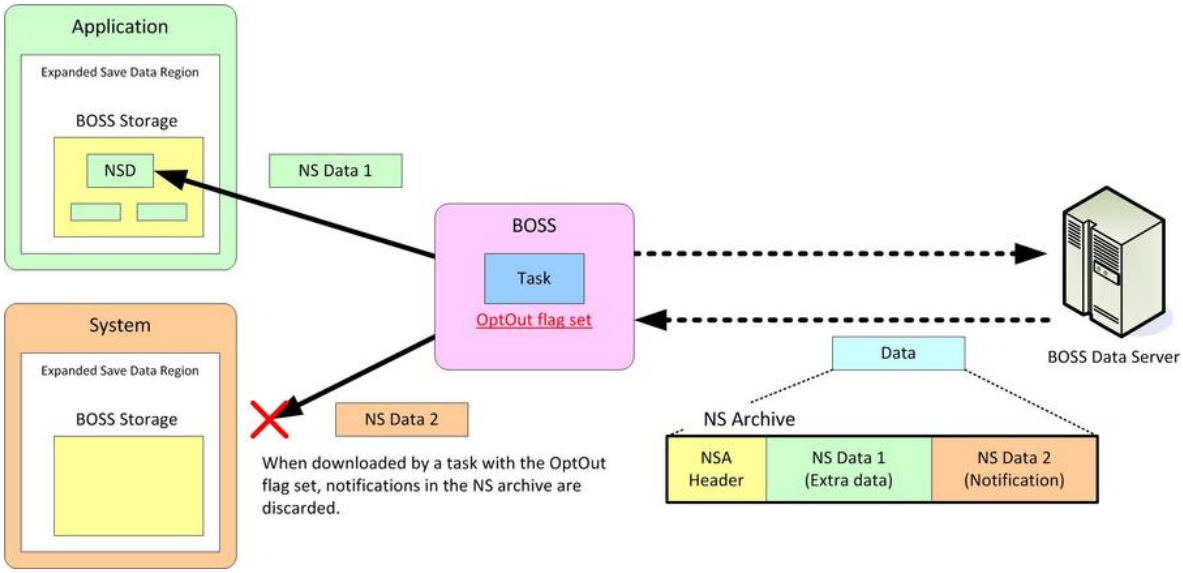
**Note:** The serial ID and version for NS data can be set on the BOSS management site. However, because the serial ID and version for Notifications and Data Store data are set automatically, there is nothing the developer can do to set them. For more information, see the manual available on the BOSS management site.

**Warning:** Serial IDs in the range from 4294901760 through 4294967295 (`0xFFFF0000` through `0xFFFFFFFF`) are reserved for the system and are prohibited from use by general applications. Use a value outside of this range (1 through 4294901759 (`0xFFFEFFFF`)) when setting a serial ID on the BOSS data server's BOSS management website.

NS data includes various properties that can be accessed by the application, such as the data's serial ID, data type, and size. However, notifications are automatically processed by the system and cannot be directly loaded by applications. Although downloaded contextual banner data is also processed automatically by the system, it can be loaded by applications.

NS archives can contain notifications. You can set the `OptOut` flag separately for each BOSS storage location to prevent such notifications from being processed and displayed in the Notifications applet.

**Figure 4-8. Processing Flow When the OptOut Flag Is Set**




Call the `nn::boss::SetOptoutFlag()` and `nn::boss::GetOptoutFlag()` functions to set and get the `OptOut` flag.

**Code 4-38. Setting and Getting the OptOut Flag**

```
nn::Result nn::boss::SetOptoutFlag(bool flag);
nn::Result nn::boss::GetOptoutFlag(bool* pFlag);
```

Call `nn::boss::SetOptoutFlag()` and pass `true` for the *`flag`* parameter to set the `OptOut` flag and ignore any notifications that might be included in downloaded NS archives. (BOSS discards any such notifications.) It has the same behavior as when opting out of receiving notifications from the application in the notification list.
Also, the current setting can be retrieved with the `nn::boss::GetOptoutFlag()` function.

## 4.2.2.2. BOSS Storage

Extra data and downloaded contextual banner data files are saved to an archive memory region called BOSS storage, which is created in the expanded save data region specified by the application. Although BOSS storage is created in the application's expanded save data region, applications cannot access it directly. Applications specify to BOSS how much of the expanded save data memory region to use for BOSS storage.

After BOSS storage is full, BOSS automatically deletes data starting from the earliest (the data with the lowest serial ID) so that downloaded data does not exceed the specified storage capacity. You can also specify the maximum number of data files when registering BOSS storage from within the application. When you attempt to save data in excess of the maximum number of data files, BOSS deletes data, starting with the earliest data first, so that the amount of data in storage never exceeds the maximum amount. The maximum number of data files defaults to 2000 unless otherwise specified. Applications must factor this in when specifying the size and number of files for BOSS storage, to accommodate both the size and number of files of downloaded data.

In addition, when BOSS storage has been newly created in expanded save data, a maximum value is set for the number of files that can be created in the expanded save data. Because the NS data files in BOSS storage also count toward the total number of files in expanded save data, make sure to set a value that takes the number of instances of NS data being saved in BOSS storage into account when setting the maximum number of files to be created in extended save data.

**Warning:** If even the deletion of earlier data would not free enough memory to store the data to be downloaded, the task is canceled and an error is generated (`NSA_ERROR_STORAGE_INSUFFICIENCY` ). This can occur in the following scenarios.

- The size of data to download exceeds the size of BOSS storage.
- The data to download is a newer version of data already downloaded, and the size of the old and new versions combined exceeds the size of BOSS storage.

This first scenario is obvious, but you must be careful about the second scenario for data that might be updated to a newer version. When you download a newer version of downloaded data, the new version of the data is first downloaded as a temporary file, the old version is then deleted, and the temporary file is renamed with the official filename. In other words, both the old and new versions temporarily exist in BOSS storage, so their combined size must be less than the size of BOSS storage.

This error does not occur if BOSS storage is greater than double the size of the maximum size of NS data.

Downloaded NS data is saved temporarily to the application's BOSS storage before the different parts of the data are saved to their final destinations. Consequently, notification data is briefly stored in the application's BOSS storage, as shown in Figure 4-7, even though it will ultimately be saved to the system's BOSS storage. If there are already *N* NS data files in a BOSS storage region and *N* is its maximum number of data files, notification data downloaded to that region may cause an existing NS data file to be deleted to make room for temporarily saving the new notification data. That temporary data could then be deleted after being saved to its final destination. Consequently, the number of NS data files in that application's BOSS storage is decreased by one (*N-1*) after downloading new notification data. If your tasks distribute or download notification data, take this behavior into account when configuring your BOSS storage.

**Note:** Notifications data that is temporarily saved in BOSS storage is immediately moved to expanded save data on the console. Notifications are not received more than once as long as the task is not removed and re-registered. Previously downloaded data remains on the system and the serial ID and version are compared.

**Warning:** If the number of files in expanded save data (including the number of instances of NS data) reaches the maximum value, NS data downloaded after that point cannot be

saved, and an error results upon execution of each task.

NS data saved to BOSS storage has a header of 52 bytes added to the front of the file. Consequently, when specifying the size of BOSS storage, make sure that you take this into consideration.

BOSS storage is only meant to serve as the BOSS working memory region, and data might be automatically deleted. Consequently, do not leave any data that you want to keep permanently as extra data in BOSS storage as NS data. Instead, we recommend loading this NS data via BOSS and saving a copy as a separate file in the expanded save data region. Also, when a lot of data is being stored in BOSS storage, routines that target all data in BOSS storage become less efficient. We also strongly recommend that you delete data that will no longer be used from BOSS storage. As described above, you can also delete data from BOSS storage if you permanently save the NS data outside BOSS storage.

When an expanded save data region is shared among multiple titles in a series, an application in that series can access downloaded data for any title in the same series.

Developers must submit an application to Nintendo (support@noa.com) to use the BOSS service. In response, Nintendo will provide dedicated URLs for each application. BOSS communication is via the HTTPS protocol using the 3DS system's built-in CA certificates.

Lotcheck will verify whether applications that use BOSS services register tasks using the settings that were specified in the application to use BOSS services that was submitted to Nintendo.

**Warning:** The NS data file serial ID must be unique within the group of files belonging to the same application (or to the same series for series data).

### Downloading Notifications Only

Because notifications are stored in the expanded save data of the system, NS data does not accumulate in BOSS storage for applications that download only notifications. However, even in such cases, you must register for BOSS storage for the amount needed to temporarily store notification data. This requirement occurs because hash and signature authentication is carried out after NS data is temporarily stored in BOSS storage. These processes must succeed first, and then NS data is moved into the BOSS storage where it will be temporarily stored. In other words, because the BOSS storage used to store the data is the same as that used to store data for applications that have registered tasks, the NS data remains behind unchanged. In the case of notification data, it is moved to the expanded save data of the system.

60 KB of BOSS storage is required in order to download the maximum amount of notification data (when using 50 KB of attachment data).

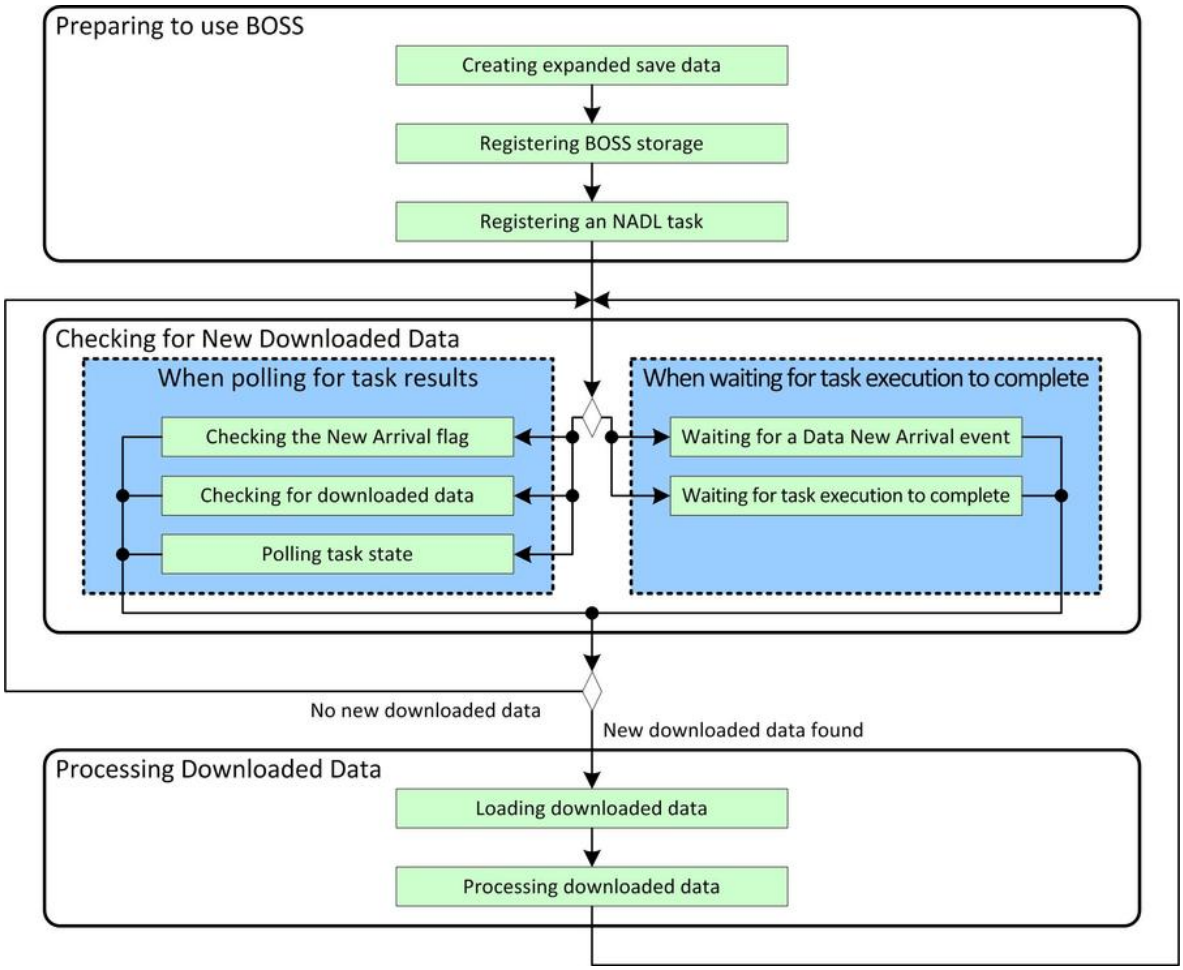
**Warning:** When using expanded save data only to distribute notifications, check at regular intervals (such as on startup) to make sure that the expanded save data can be mounted normally. Downloaded data is temporarily saved to the expanded save data

region, and if the expanded save data is corrupted or deleted, notifications will no longer be received, even if the task still exists.

## 4.2.2.3. Required Steps to Process an NADL Task

The following figure shows the required steps from registering an NADL task using BOSS through processing the downloaded data.

**Figure 4-9. Required Steps to Process an NADL Task**


Preparing to use BOSS
Creating expanded save data
Registering BOSS storage
Registering an NADL task
Checking for New Downloaded Data
When polling for task results
Checking the New Arrival flag
Checking for downloaded data
Polling task state
When waiting for task execution to complete
Waiting for a Data New Arrival event
Waiting for task execution to complete
No new downloaded data
New downloaded data found
Processing Downloaded Data
Loading downloaded data
Processing downloaded data

The following table provides an overview of each step.

**Table 4-18. Overview of NADL Task Processing**

| Processing Step | Description |
|---|---|
| Create Expanded Save Data | The application must first create an expanded save data region before it can register BOSS storage in which to save data downloaded by NADL tasks.<br>For more information about creating expanded save data regions, see the *3DS Programming Manual: System*. |
| Register | |

| | |
|---|---|
| BOSS Storage | Register the BOSS storage for saving the data downloaded by an NADL task.<br>For more information, see 4.2.2.4. Preparing to use BOSS. |
| Register an NADL Task | Configure the actions for an NADL task and register the task with BOSS.<br>For more information, see 4.2.2.5. Registering a Nintendo Archive Download (NADL) Task. |
| Check the New Arrival Flag | Check the new arrival flags managed by the library.<br>For more information, see 4.2.2.6. Checking the New Arrival Flag and Waiting for a Data New Arrival Event. |
| Check for Downloaded Data | Get a list of the downloaded data and check whether there is any newly downloaded data. For already-processed data, the application must handle any other operations as necessary: deletion, serial ID recording, setting the data's status from unread to read, and so on.<br>For more information, see 4.2.2.7. Checking for Downloaded Data. |
| Poll Task State | Poll the state of the NADL task to determine whether the state is "completed."<br>For more information, see 4.2.1.5. Task Information. |
| Wait for a Data New Arrival Event | Register an event with the library and wait for that event to enter the signaled state.<br>For more information, see 4.2.2.6. Checking the New Arrival Flag and Waiting for a Data New Arrival Event. |
| Wait for Task Execution to Complete | Run the NADL task in the foreground and wait for it to complete.<br>For more information, see 4.2.1.4. Registering and Running Tasks. |
| Load Downloaded Data | Access the NS data saved in BOSS storage and load it into a buffer allocated by the application.<br>For more information, see 4.2.2.8. Loading Downloaded Data. |
| Process Downloaded Data | Process the obtained downloaded data in the application. |

## 4.2.2.4. Preparing to Use BOSS

BOSS storage must already be registered before you can register and run NADL tasks.

**Code 4-39. Registering and Unregistering BOSS Storage**

```
nn::Result nn::boss::GetStorageInfo(size_t* pStorageSize=NULL);
nn::Result nn::boss::GetStorageInfo(size_t* pStorageSize, u16* pEntryCount);
nn::Result nn::boss::RegisterStorage(
        bit32 storageId, size_t storageSize,
        nn::boss::StorageType storageType=nn::boss::USER_EXT_SAVEDATA_STORAGE);
nn::Result nn::boss::RegisterStorage(
        bit32 storageId, size_t storageSize, u16 entryCount,
        nn::boss::StorageType storageType=nn::boss::USER_EXT_SAVEDATA_STORAGE);
nn::Result nn::boss::UnregisterStorage(void);
```

Before registering BOSS storage, check whether BOSS storage has been created by calling `nn::boss::GetStorageInfo`. If it has not yet been created, the function returns

`nn::boss::ResultStorageNotFound`. For *`pStorageSize`* and *`pEntryCount`*, specify pointers to variables that store, respectively, the size and maximum number of data files for the BOSS storage.

Call `nn::boss:RegisterStorage` to create BOSS storage in the expanded save data memory region specified by *`storageId`*. This created BOSS storage can store up to an all-files combined total size of *`storageSize`*. Use the overloaded version that has an *`entryCount`* parameter to specify the maximum number of data files that can be registered to the BOSS storage.

Call `nn::boss:UnregisterStorage` to unregister BOSS storage. Attempting to register BOSS storage in an expanded save data region to which the application does not have access permission returns an `nn::boss::ResultStorageAccessPermission` error.

Each application can only register one BOSS storage area. To change the expanded save data region used as BOSS storage by an application, first call `nn::boss::UnregisterStorage` to unregister, and then call `nn::boss::RegisterStorage` again to create a new BOSS storage area.

**Warning:** When registered tasks remain after unregistering BOSS storage with `nn::boss::UnregisterStorage()`, the following behavior occurs.

- Tasks are executed without BOSS storage, and download data is discarded. When executing subsequent tasks, there may be processing to prevent duplicate downloads.
  Even if the application is not receiving distribution data, as long as processing to prevent duplicate downloads is being carried out, the corresponding distribution data cannot be received. Tasks can be added and downloaded once again.
- Even when BOSS storage is added again, existing tasks continue to use the previous BOSS storage size.
  If tasks from the previous BOSS storage size and tasks from the new BOSS storage size are mixed up, the data with sizes different from the current BOSS storage size might be replaced and deleted.

Because of these limitations, we strongly recommend deleting all tasks when deleting BOSS storage.

You can use the following procedure to safely change the BOSS storage size.

1. Unregister all tasks.
2. Unregister BOSS storage.
3. Register BOSS storage with the new size.
4. Re-register tasks.

**Warning:** After changing the expanded save data memory region used for BOSS storage, you can no longer access any NS data saved in the previous BOSS storage area.

We recommend creating BOSS storage in one expanded save data region and keeping

it there.

**Notes:** For more information about accessing the expanded save data memory region, see the *3DS Programming Manual: System*.

For more information about notification data, see the *Applet Specification*. For more information about downloaded contextual banner data, see Banner Overview and ctr_makedlexbanner.

#### Handling the Deletion of Expanded Save Data by Users

Note the possibility that a user may delete expanded save data in System Settings. If such data is deleted, the application must re-create the expanded save data to use BOSS again. An application that uses BOSS must always be able to handle the deletion of expanded save data by users. Applications are required to implement some process for handling this situation. One example for handling this is to check for expanded save data when the application starts, and if none exists, prompt for user confirmation to create expanded save data.

#### Maximum Number of BOSS Storage Regions That Can Be Registered and Automatic Unregistering

Up to 127 BOSS storage regions can be registered. This number represents the total number of BOSS storage regions that can be registered by all applications, including built-in applications. If an application attempts to register BOSS storage when the maximum number has already been reached, BOSS automatically unregisters the BOSS storage for some application with no registered tasks and registers the new BOSS storage. If there are no applications with zero registered tasks, the product of the execution count and the execution interval is calculated for each task, and those tasks with the smallest resulting values are unregistered until some application ends up with zero tasks. The BOSS storage for that application is then unregistered the new BOSS storage is registered.
If an application has not been run for a long time, its previously registered BOSS storage might get unregistered by BOSS using this automatic unregistering process. No tasks are executed at that time because registered tasks are also canceled. For this reason, applications that use BOSS must check whether BOSS storage has been registered when they start.

## 4.2.2.5. Registering a Nintendo Archive Download (NADL) Task

Use the `nn::boss::NsaDownloadAction` class to set the actions for an NADL task. Create an instance of the class and then call `Initialize` to set its URL property.

**Code 4-40. Initializing the nn::boss::NsaDownloadAction Class**

```
class nn::boss::NsaDownloadAction : public nn::boss::TaskAction
{
    nn::Result Initialize(const char* pUrl);
}
```

For *pUrl*, specify the URL of the NS archive to download. This URL value can be up to 512 characters, including the NULL terminator.

The URL for the NS archive takes the following format.

> https://<CDN domain>/<environment ID>/<common parameter>/<BOSS code>/<task ID>/<optional filename>

To distribute NS archives in different languages, you must use a format that contains language information (ISO 639-1 alpha-2).

> https://<CDN domain>/<environment ID>/<common parameter>/<BOSS code>/<task ID>/<language information>/<optional filename>

**To distribute NS archives in different languages and also to use both simplified and traditional Chinese characters in notifications, you must use a format that contains not only language information but also country information (ISO 3166-1 alpha-2).**This is because simplified and traditional Chinese are both defined as *Chinese* (zh) in ISO 639-1 alpha-2, so the language information alone is not enough for the BOSS data server to determine which character set to use for the distribution.

> https://<CDN domain>/<environment ID>/<common parameter>/<BOSS code>/<task ID>/<country information>/<language information>/<optional filename>

**Warning:**If the user has changed the language setting or country setting in System Settings and tasks are not reregistered, a task that is distributed in different languages will download data in a language different from what is set in System Settings.

**Note:**You must apply on OMAS to use the feature to distribute in different languages. If you intend to distribute using both simplified and traditional Chinese characters, indicate that fact in the relevant field on the application form.

Systems set to the China region (mainland China) display notifications using simplified Chinese characters, whereas systems set to the Taiwan region (Taiwan/Hong Kong) display notifications using traditional Chinese characters.

The `NsaDownloadAction` class inherits the `nn::boss::TaskAction` class and also has member functions for setting common task actions.

**Table 4-19. Valid Property IDs for the nn::boss::NsaDownloadAction Class**

| Property ID | Type | Property |
|---|---|---|

| | | |
|---|---|---|
| ACTION_URL | char[MAX_URL_LENGTH] | Destination URL. |
| ACTION_HTTP_HEADER | struct httpRequestHeader[MAX_HTTP_HEADER] | Additional HTTP request header. |
| ACTION_CLIENT_CERT | u32 | Built-in client certificate ID. |
| ACTION_ROOT_CA | u32 | Built-in root CA certificate ID. |
| ACTION_AP_INFO | ApInfoType | AP information added to the HTTP request. |
| ACTION_CLIENT_CERT_NUM | u32 | Number of built-in client certificates to register. |
| ACTION_ROOT_CA_NUM | u32 | Number of built-in root CA certificates to register. |

Unique NADL task action settings are described below.

#### Usable Root CA Certificates

The library automatically registers the NADL task's root CA certificate, so the application does not need to do this.

The library registers two root CA certificates, specifically a root CA certificate used to connect to the BOSS data server for live distribution to retail products, and a root CA certificate used to connect to the BOSS data server for test distribution. Data for live distribution can only be downloaded from the BOSS data server using the retail (production) hardware. However, an exception is made for the library sample demos, and so data for live distribution can also be downloaded using the development hardware. Data for test distribution can be downloaded using both the development hardware and retail systems, if you specify a special DNS server in the System Settings. You can check the IP address of this DNS server by performing a test distribution with the BOSS Management Tool of the BOSS data server.

## 4.2.2.6. Checking the New Arrival Flag and Waiting for a Data New Arrival Event

Use the following functions to check whether any new data has been downloaded.

**Code 4-41. Checking for New Data**

```
nn::Result nn::boss::GetNewArrivalFlag(bool* pFlag);
nn::Result nn::boss::RegisterNewArrivalEvent(nn::os::Event* event);
```

The `nn::boss::GetNewArrivalFlag()` function returns `true` in its *`pFlag`* parameter if there is new data.

The `nn::boss::RegisterNewArrivalEvent()` function signals the event class passed in its *`event`* parameter when new data is downloaded. The application must initialize this event class in advance.

> **Note:** Any downloaded data is not treated as new data if the NS archive's new arrival flag is cleared. After downloading such data, the `GetNewArrivalFlag()` function continues to return `false`, but the event passed to `RegisterNewArrivalEvent` is signaled.

## 4.2.2.7. Checking for Downloaded Data

After the task has finished running, the downloaded NS data is saved to the BOSS storage area of the expanded save data region specified in the task. You must first get a list of NS data serial IDs before you can access the NS data saved to BOSS storage.

**Code 4-42. Getting a List of NS Data Serial IDs**

```
nn::Result nn::boss::GetNsDataIdList(
                        u32 dataType, nn::boss::NsDataIdList* pNsDataId);
nn::Result nn::boss::GetNewDataNsDataIdList(
                        u32 dataType, nn::boss::NsDataIdList* pNsDataId);
```

Call the `nn::boss::GetNsDataIdList()` and `nn::boss::GetNewDataNsDataIdList()` functions to get a list of the NS data files that the application can load. `GetNsDataIdList` gets a list of all such data in the descending order of the serial IDs, and `GetNewDataNsDataIdList()` gets a list of only unread data.

The update indicator displayed on the application's icon is cleared after calling either of these functions (that is, the new arrival flag obtained by `nn::boss::GetNewArrivalFlag` is set to `false`). Newly downloaded NS data is marked as unread. Call the `nn::boss::NsData::SetReadFlag()` function to mark NS data as either read or unread. NS data is left as unread unless this setting is changed by the application.

> **Note:** When using the list of serial IDs to determine whether downloaded data must be processed, your application must identify the processed data in some way, such as by always deleting the data after processing, recording the serial IDs of processed data, or setting the processed NS data to read status.
>
> When multiple titles in a series share a single expanded save data region, NS data downloaded via a task registered by a series title can be accessed by other titles in that same series. For more information, see Sharing NS Data.
>
> Because these functions search through all data in BOSS storage, the more data stored in BOSS storage, the more time execution requires. To maintain efficiency of

execution, we recommend that you regularly delete data in BOSS storage that is no longer being used.

For each of these functions, specify the NS data search conditions in the *dataType* parameter, and specify a pointer to an instance of the `nn::boss::NsDataIdList` class to store the NS data serial ID list in the *pNsDataId* parameter.

Specify the NS data search conditions as follows.

Specify `DATA_TYPE_ALL` to include all NS data in the search targets.

Otherwise, specify the search targets using a bitwise OR of the global data type and private data type. This stores on the list only NS data IDs that have a matching global data type and a private data type that results in a match when masked.

The global data types fall into three categories, as shown in the following table.

**Table 4-20. NS Data Types**

| Definition | Description |
|---|---|
| `DATA_TYPE_APPDATA` | Extra data. |
| `DATA_TYPE_NEWS` | Data displayed in the notification (the application cannot get this data). |
| `DATA_TYPE_EXBANNER` | Downloaded contextual banner data. |
| `DATA_TYPE_DATASTORE` | Data obtained with a Data Store download task. |

The private data type is a 16-bit value and is used as the mask value for NS data (the least-significant 16 bits of the data type). NS data that has even one bit set to 1 that matches the corresponding bit in the specified private data type (in other words, bits where a bitwise AND of the two private data type values is not `0`) is included in the list. For example, to include all extra data in the serial ID list, set the *dataType* parameter to a value of `DATA_TYPE_APPDATA | 0x0000FFFF`. Applications can use NS data types to get extra data that is sorted into custom categories.

**Note:** If BOSS storage cannot be accessed normally—for example, because the SD card has been removed—the `nn::boss::GetNsDataIdList()` function returns an `nn::boss::ResultFileAccess` error. The application cannot recover from this situation, so you must display the error code to the user. The BOSS sample demo `sample_nadl_simple` (CTR-SDK 7.1 and later) provides sample implementation of the display of error codes for this error and other unanticipated errors.

To instantiate the `nn::boss::NsDataIdList` class, you need a `u32` array to store the serial IDs. The array must be big enough to contain all the serial IDs that can be obtained in a single search.

The search function returns `nn:boss::ResultNsDataListSizeShortage` if the array is not big enough to hold all of the serial IDs found in a search. If this happens, you can search again

using the same instance (run the same function again) to get a list of serial IDs starting where the previous execution left off.

The search function returns `nn::boss::ResultNsDataListUpdated` if new NS data is added to BOSS storage while searching. If this happens, first call the `Initialize` member function of the `nn::boss::NsDataIdList` class instance to initialize the list, and then search to get the serial ID list starting again from the beginning.

The `IsSuccess` method on the `nn::Result` instance returned by the search function returns `true` after it has listed all NS data files matching the search terms.

**Code 4-43. nn::boss::NsDataIdList Class**

```
class nn::boss::NsDataIdList
{
    explicit NsDataIdList(u32* pSerial, u16 size);
    virtual ~NsDataIdList(void);
    void Initialize(void);
    u16 GetSize(void);
    u32 GetNsDataId(u16 index);
}
```

You must call `Initialize` after generating an instance. Call `Initialize` on an instance that you have already used for a search to reset the search to get the list of serial IDs again from the beginning.

Call `GetSize()` to get the number of serial IDs obtained.

Call `GetNsDataId()` and specify the array index to get a serial ID. The function returns `INVALID_SERIAL_ID` if the specified index is out of bounds.

### Sharing NS Data

When multiple titles in a single series share an expanded save data region, any application in that series can call `nn::boss::GetNsDataIdList` or `nn::boss::GetNewDataNsDataIdList` to search the NS data downloaded by any application in that series, and use the `nn::boss::NsData` class to access that NS data. In other words, NS data downloaded by an NADL task registered by one series title is shared among all titles in that series. NS data for one title in a series is shared among all titles in that series.

> **Warning:** Consequently, NS data serial IDs must be unique not just among files downloaded by that application but also among all NS data files downloaded by any application in that whole series.

If you do not need to work with NS data downloaded by other titles in a series, you can call the `nn::boss::GetOwnNsDataIdList` or `nn::boss::GetOwnNewDataNsDataIdList()` functions to search for and list just the NS data downloaded by tasks registered by the calling application.

**Code 4-44. Getting a List of NS Data Serial IDs for Just the Calling Application**

```
nn::Result nn::boss::GetOwnNsDataIdList(
                        u32 dataType, nn::boss::NsDataIdList* pNsDataId);
nn::Result nn::boss::GetOwnNewDataNsDataIdList(
                        u32 dataType, nn::boss::NsDataIdList* pNsDataId);
```

These two functions differ from `nn::boss::GetNsDataIdList()` and `nn::boss::GetNewDataNsDataIdList()` only in which NS data files they search for. They otherwise operate identically. However, note that these functions incur an increased processing load because they check if an NS data file was downloaded via a task registered by the calling application.

## 4.2.2.8. Loading Downloaded Data

Access NS data by generating an instance of the `nn::boss::NsData` class and calling `Initialize` using the obtained serial ID as an argument.

**Code 4-45. nn::boss::NsData Class Initialize and Read Functions**

```
class nn::boss::NsData
{
    nn::Result Initialize(u32 serial);
    nn::Result GetHeaderInfo(HeaderInfoType type, void* pValue, size_t size);
    nn::Result SetReadDataPosition(s64 position, PositionBase base);
    s32 ReadData(u8* pDataBuf, size_t bufLen);
    nn::Result SetReadFlag(bool flag);
    nn::Result GetReadFlag(bool* pFlag);
}
```

NS data properties can be obtained from the header information returned by the `GetHeaderInfo()` function. The header information type in the *`type`* parameter specifies which header information is obtained. For the *`pValue`* and *`size`* parameters, specify a variable to hold the header information and its size in bytes (which depends on the data type of the header information).

**Table 4-21. Header Information Types**

| Header Information Type | Type | Description |
|---|---|---|
| NSD_TITLEID | s64 | Title ID. |
| NSD_FLAGS | u32 | NSD flag (not used). |
| NSD_DATATYPE | DataType | Data type. |
| NSD_LENGTH | u32 | Data length. |
| NSD_SERIALID | u32 | Serial ID. |
| NSD_VERSION | u32 | Version number. |

Call the ReadData() function to load (read) the NS data. Specify a read buffer in *pDataBuf* and its size in *bufLen*. The function returns the number of bytes read. A return value of 0 indicates that the function has already read all the data. An error occurs and the function returns a negative number if it attempts to read data that does not exist, which could happen if data was automatically deleted to allocate a save region. An error occurs and the function returns NN_BOSS_NSDATA_READ_ERROR_UPDATED if the data is overwritten by new download data while it is being read.

Call the SetReadDataPosition() function to specify where in the NS data to read (equivalent to the file system Seek() function).

> **Warning:** Like the file system Seek() function, NS data read operations are substantially slower when the read position set by the SetReadDataPosition() function is not a multiple of 4 from the start of the file.

Call the SetReadFlag() function specifying true for the *flag* parameter to mark NS data as read. Call the GetReadFlag() function to check whether NS data has been read. The function returns a value of true in the *pFlag* parameter if the NS data is marked as read.

You can also use the following functions to work with NS data.

**Code 4-46. Setting and Getting Additional Information, and Deleting Data**

```
class nn::boss::NsData
{
    nn::Result SetAdditionalInfo(u32 info);
    nn::Result GetAdditionalInfo(u32* pInfo);
    nn::Result GetLastUpdated(nn::fnd::DateTime* PTime);
    nn::Result Delete(void);
}
```

Call the SetAdditionalInfo() and GetAdditionalInfo() functions to set and get additional information as a u32 value that applications can freely add to NS data. Note that this additional information is lost when the NS data is overwritten with a new version.

Call the GetLastUpdated() function to get the date and time when an NS data file was created. This indicates the last time the NS data was overwritten with a new version.

Call the Delete() function to delete NS data from BOSS storage. Even without explicitly calling the Delete() function, the BOSS daemon automatically frees memory by deleting data starting from the earliest (the data with the lowest serial ID) after the storage area is full (that is, after the total size of the NS data exceeds the size specified to the nn::boss::RegisterStorage() function). You can explicitly call the Delete() function to get rid of unneeded NS data and keep the BOSS daemon from having to manage that data, thereby improving processing efficiency. Deleted data, for example, is no longer processed when searching NS data to get serial ID lists, making for more efficient searching.

### Preventing Duplicate Downloads

BOSS uses the `If-Modified-Since` HTTP request field to keep from downloading the same NS archive multiple times.

However, the same NS archive is downloaded multiple times if the following conditions are all fulfilled simultaneously.

- After previously downloading the NS archive, 50 or more other NS archives have been downloaded in the same task.
- A newer NS archive is registered on the server than the last downloaded NS archive. In other words, the update time of the NS archive on the server is more recent than the `LastModifiedTime` of the last downloaded NS archive.

There is no need for applications to build in a way to track this information, but each NS archive is assigned its own ID, which is used to control NS archive downloads to prevent downloading duplicates. Each task can only record the last 50 IDs, so distributing an NS archive after a system has downloaded 50 other NS archives can result in an archive being downloaded again. However, so long as the `LastModifiedTime` of the NS archive is not updated to be more recent than the last downloaded NS archive, the `If-Modified-Since` field prevents that archive from being downloaded again.

Even when these conditions are not fulfilled during regular use, data write operation errors or other unexpected bugs on the BOSS data server can potentially cause the same NS archive to be downloaded multiple times. Consequently, make sure that such a situation does not cause any fatal errors in your application. For example, an application could keep track of the serial IDs of the data already obtained, and then delete any earlier data if data with the same serial ID is obtained again later.

> **Warning:** Be cautious when a task has been deleted and re-registered. In general, functionality to prevent multiple receives does not work because previously downloaded information has already been deleted.
>
> To prevent the repeated receipt of notifications, implement programming such that tasks that receive notifications do not need to be deleted. For example, avoid an implementation that deletes and reregisters tasks every time the application starts.

### Interrupted Downloads

The download will be interrupted if the user turns the power off, the battery becomes low, or the connection to the access point is lost due to a poor communication environment. Although the system ordinarily tries the download again after 10 minutes, if the communication environment is poor, the system may not always be able to retry before the task is executed again.

### Sleep Mode

Although an ongoing download will be interrupted if the system goes into Sleep Mode, the system

will resume the download as soon as it finds an access point. After returning from Sleep Mode, if the system connects to the same access point as before it went into Sleep Mode, it continues the download where it left off. If the system was offline before entering Sleep Mode (but was connected during Sleep Mode), it disconnects from the access point but resumes the download the next time that it connects to an access point.

#### NS Data That Has Been Corrupted or Tampered With

Use the following procedure to check whether NS data has been corrupted or tampered with.

1. Get the NS data ID with the `nn::boss::GetNsDataIdList()` function.
2. Get an illegal-use error with the `nn::boss::NsData::GetHeaderInfo` or `nn::boss::NsData::ReadData()` function.
   - If the `GetHeaderInfo` function returns an error other than `ResultInvalidNsDataValue`, `ResultIpcNotSessionInitialized`, or `ResultStorageAccessPermission`, the NS data has been corrupted.
   - If the `ReadData()` function returns `NsData::NN_BOSS_NSDATA_READ_ERROR_READ_DATA`, the NS data has been corrupted. If it returns `NsData::NN_BOSS_NSDATA_READ_ERROR_GET_HEADER`, get the error details using the `GetHeaderInfo()` function to determine whether the NS data has been corrupted or tampered with.

Perform the following when corruption or tampering is detected in NS data.

1. Delete NS data that has been corrupted or tampered with using the `nn::boss::NsData::Delete()` function.
2. Notify the user that data has been corrupted and instruct the user to immediately delete the data if a notification has arrived at the same time, or when there has already been advance notice of arrival of data in the application.

As a rule, because of the feature to prevent duplicate downloads, the system cannot download the same archive for the same task again (unless all the right conditions are present), even when NS data is deleted. To download deleted NS data again, first cancel registration for the task using the `nn::boss::UnregisterTask()` function, and then use the `nn::boss::RegisterTask()` function to reregister the task. Note that data downloaded may differ from the data that was deleted if NS data has been updated on the server. Keep this in mind when implementing the re-download process.

## 4.2.3. NSA List

The NSA list is a feature for retrieving listings of the NS archives on the BOSS data server. You can perform searches by specifying the BOSS code assigned when requesting the use of SpotPass, the task ID, or other attributes. A list of NS archive records matching the conditions is returned.

**Note:** For more information about the settings methods for BOSS code and the NSA List, see

OMAS Help.

## 4.2.3.1. NsaList Class

NSA list features are defined in the `nn::boss::NsaList` class.

**Code 4-47. Constructor and Destructor for the NsaList Class**

```
class nn::boss::NsaList
{
    NsaList(const char* nsaListFilePath);
    ~NsaList(void);
}
```

The path for storing the retrieved NSA list file is specified in the *nsaListFilePath* parameter of the constructor. Specify a path within an archive that can be written by the application, as with expanded save data.

## 4.2.3.2. Getting an NSA Listing

Use the `Download()` function to get the NSA listing for a specific BOSS code and task ID. The function is executed in the foreground as an immediate task, so a connection to the Internet must be established using the AC library in advance.
When getting this information, the country and language information set in System Settings is automatically embedded in the query string.

**Code 4-48. Function for Getting an NSA List**

```
nn::Result nn::boss::NsaList::Download(
        const char* bossCode, const char* taskId,
        const nn::boss::NsaList::SearchAttributes* attributes = NULL,
        u32 waitTimeoutSec = NN_BOSS_NSALIST_WAIT_FINISH_TIMEOUT_DEFAULT_SEC,
        s64 fileSize = NN_BOSS_NSALIST_MAX_SIZE);
nn::Result nn::boss::NsaList::Cancel(void);
```

Specify the BOSS code in *bossCode* and the task ID in *taskId*.

Specify a download timeout time in seconds in *waitTimeoutSec*. If this parameter is omitted, `nn:: boss::NN_BOSS_NSALIST_WAIT_FINISH_TIMEOUT_DEFAULT_SEC` (currently 30 seconds) is specified. The call does not time out if a value of zero is specified.

Search conditions are specified using three attributes (attribute 1 to attribute 3; each attribute has an ASCII string of up to nine characters) in the *attributes* parameter. NS archive entries are retrieved if the entry attributes are the same as those specified in the call to the `Download()` function. Each attribute is handled by sequence, not by combination. If attribute 1 retrieves the

NSA list of the archive called "item," even if "item" is specified in attribute 2 or attribute 3, the archive is not a target for listing unless attribute 1 is not "item." If the NSA list is retrieved with conditions of attribute 1 = "A," attribute 2 = "B," and attribute 3 = "C," all of the conditions are connected by a logical AND, and only the archive that matches the sequences for all attributes is the target for listing. Entries matching these attributes are retrieved, and entries without them (NULL or empty string) are not. However, archive entries that do not specify an attribute that is specified in a `Download` call are not included in the NS archive list.

As shown in the following table, a decision is made for each of the three attributes. If the decision in all three cases is to get the archive entry, it is added to the list.

**Table 4-22. Decision to Get Based on Specified Attributes**

| Attribute in Archive | Attribute in Download Call | Get Decision |
|---|---|---|
| Not specified | Not specified | Get |
| Not specified | Specified | Do not get |
| Specified | Not specified | Get |
| Specified | Specified | Get if values match |

The attributes specified in the `Download` call are the conditions used to extract data when creating the list. Only one condition can be specified per attribute. To specify more than one condition for a single attribute, the list must be retrieved without that attribute and the content of the list parsed later.

In *`fileSize`*, specify the maximum size of the NSA List that is to be created. After creating a blank file of this size, the NSA List feature writes the NS archive list acquired by the file. When omitted, the value for the maximum size of the NSA List saved in `nn::boss::NN_BOSS_NSALIST_MAX_SIZE` is set automatically.

The NSA List size is calculated using the following formula.

NSA List Size = 54 + 236 * number of NS archives included in the list

Lists including a maximum of 1000 pieces of NS archive information are acquired, so the maximum size of an NSA List is 236,054 bytes. This value can also be set explicitly only when it is not expected that lists with a maximum number of NS archive information will be acquired, or when you want to reduce the capacity of files allocated for NSA lists.

The `Download` process can be canceled with the `Cancel()` function. `Cancel` requests are processed asynchronously, so calling this function always succeeds.

## 4.2.3.3. Checking the Retrieved Results

The `GetResult()` function can be used to confirm whether the NSA list has been retrieved.

**Code 4-49. Functions to Check the Retrieval Result**

```
nn::boss::TaskResultCode nn::boss::NsaList::GetResult(void);
u32 nn::boss::NsaList::GetHttpStatusCode(void);
```

The value returned by the `GetResult()` function is the execution result from the task used to download the NSA list. A value of `TASK_SUCCESS` indicates that retrieval of the list has completed.

If `GetResult` returns `HTTP_ERROR_*` or `SSL_ERROR_*`, a communications error has occurred. You can get the HTTP status code returned from the server using the `GetHttpStatusCode()` function. You can also use this to get detailed information about the status when an error occurs.

## 4.2.3.4. Checking Validity and Updates

Use the `CheckValidity()` function to check the validity of the downloaded NSA list. Use the `GetDigest()` function to get a digest value that can be used to confirm updates to the NSA list.

**Code 4-50. Functions to Check Validity and Updates**

```
bool nn::boss::NsaList::CheckValidity(void* pWorkBuf, size_t workBufSize);
nn::Result nn::boss::NsaList::GetDigest(u8* pDigestBuf, size_t digestBufSize);
```

Specify a working buffer and its size in *`pWorkBuf`* and *`workBufSize`*. The size of the working buffer must be at least 256 bytes. Also, when the retrieved NSA list is large, processing may be faster if the size allocated for the buffer is larger. When the `CheckValidity()` function returns `false`, the list content may have been corrupted.

The digest value obtained by the `GetDigest()` function can be used to confirm NSA list updates. The digest value is a 40-character alphanumeric string, so the buffer specified with *`pDigestBuf`* and *`digestBufSize`* must be at least 41 bytes.

If there have been no updates to the list obtained specifying the same attributes, the digest value will be the same. It can be used to determine whether the entire list has changed, but not whether individual NS archive entries have changed.

## 4.2.3.5. Parsing the NSA List

The `Parse()` function can be used to parse a retrieved NSA list.

**Code 4-51. Function for Parsing an NSA List**

```
nn::boss::NsaList::ParseResult nn::boss::NsaList::Parse(
        u32* pOutParseCount,
        nn::boss::NsaList::NsaInformation pNsaInformationArray[],
        u32 nsaInformationArraySize,
        void* pWorkBuf, size_t workBufSize, u32 nsaFirstPos = 0);
```

*pNsaInformationArray* and *nsaInformationArraySize* specify an array of NsaInformation structures that store the parsing results and the number of elements in the array. The number of list entries stored in the array is stored in *pOutParseCount*.

Specify a working buffer used for parsing and its size in *pWorkbuf* and *workBufSize* respectively. The size of the working buffer must be at least 256 bytes. Also, when the retrieved NSA list has many elements, processing can be faster if the size allocated for the buffer is larger.

When the return value is zero, all of the list entries have already been parsed. A negative return result indicates that an error has occurred. A positive result indicates that the *pNsaInformationArray* was too small, and all entries could not be parsed in one call. In this case, the Parse() function can be called repeatedly, passing the previous return result in *nsaFirstPos* to continue parsing.

The NsaInformation structure is defined as follows.

**Code 4-52. NsaInformation Structure Definition**

```
struct nn::boss::NsaList::NsaInformation
{
    char fileName[32];
    u32  fileSize;
    u32  updateEpochTime;
    char attribute1[10];
    char attribute2[10];
    char attribute3[10];
    u8   caption[150];
};
```

NS archive information is stored in each of these members.

**Table 4-23. Information Stored in Members of the NsaInformation Structure**

| Member | Stored Information |
|---|---|
| fileName | Filename. |
| fileSize | File size. |
| updateEpochTime | Update time (Number of seconds since 1970-01-01 00:00:00 GMT). |
| attribute1 – attribute3 | Contents of attributes 1 to 3 set in the archive. |
| caption | Description (UTF-8 encoded, up to 50 characters). |

## NS Archive URL

The URL specified for the NADL task has the following structure.

https://npdl.cdn.nintendowifi.net/p01/nsa/(BOSS code)/(Task ID) /(File name)?(query string)

There is an "lm" in the query string parameter. If an archive is retrieved using a NADL task with an update time (the updateEpochTime value) specified in this parameter and the update time is different, a value of 404 (Not Found) is returned. This is specified to avoid producing

inconsistencies between the content retrieved in the NSA list and the archive content during the period when the NS archive is being replaced and the contents of the CDN cache and the NS archive are not the same.

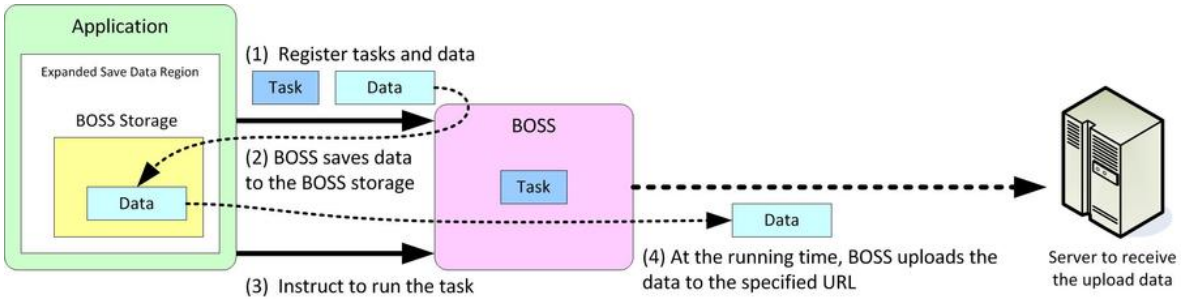
#### Use for Cache Control

The NS archive cache can be controlled easily by comparing the `updateEpochTime` member content in the `NsaInformation` structure saved from the previous time it was retrieved with the current value. However, the time measured by the 3DS system can change due to error or user action, so update times must be compared using two `updateEpochTime` values.

## 4.2.4. Data Upload Tasks

Data upload tasks are tasks in which data uploads are performed via an HTTP/HTTPS connection to an arbitrary data server.

**Figure 4-10. Uploading With a Data Upload Task**




The data upload task runs the upload as long as there are execution counts remaining. However, after the upload is successful, the same data will not be uploaded again until the task is re-registered.
When you want to upload data that is different from the data registered with a task, you must register a new task, because a task cannot be reused.

The new task can then be registered. Implement so that the task executes (the start instruction) immediately after registration. If power is interrupted while registering a task, the information for the task and its data become invalid. Tasks that could not begin due to a power interruption can be identified when the `nn::boss::Task::GetState()` function returns `TASK_REGISTERED`. To re-register a task, first delete the task if it falls under the `UnregisterTask()` function.

> **Note:** You must contact support@noa.com before using data upload tasks.

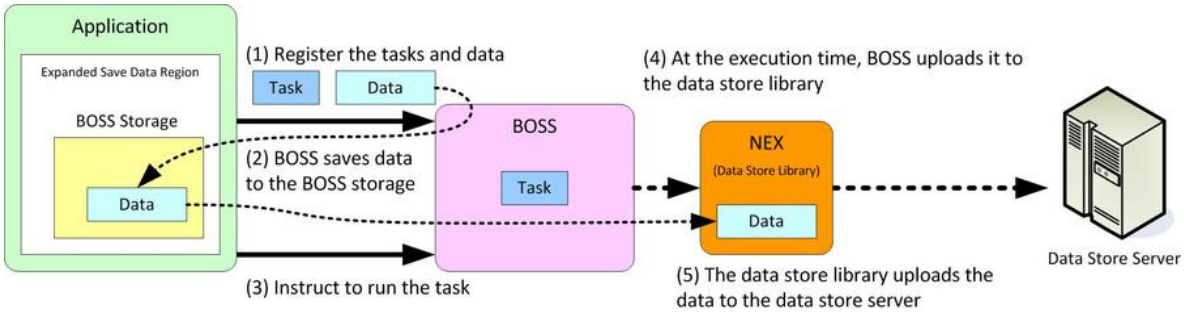
## 4.2.5. Data Store Upload Tasks

Data Store upload tasks are tasks that upload data using the NEX library's data store features.

Users can be specified, allowing them to view data that has been uploaded. Users can also exchange data among themselves.

A Data Store upload task can use the NEX library data store features, even when the application is not running (such as when it is in Sleep Mode) by communicating via SpotPass.

**Figure 4-11. Uploading With a Data Store Upload Task**




The DataStore upload task runs the upload as long as there are execution counts remaining. However, after the upload is successful, the same data will not be uploaded again until the task is re-registered.
When you want to upload data that is different from the data registered with a task, you must register a new task, because a task cannot be reused.

The new task can then be registered. Implement so that the task executes (the start instruction) immediately after registration. If power is interrupted while registering a task, the information for the task and its data become invalid. Tasks that could not begin due to a power interruption can be identified when the `nn::boss::Task::GetState()` function returns `TASK_REGISTERED`. To re-register a task, first delete the task if it falls under the `UnregisterTask()` function.

There is a maximum time lag of 20 seconds before data that has been uploaded can be downloaded.

> **Note:** This feature uses the NEX library, but you do not need to include the NEX library in the application.
>
> For more information about data store features, see the NEX Library documentation.

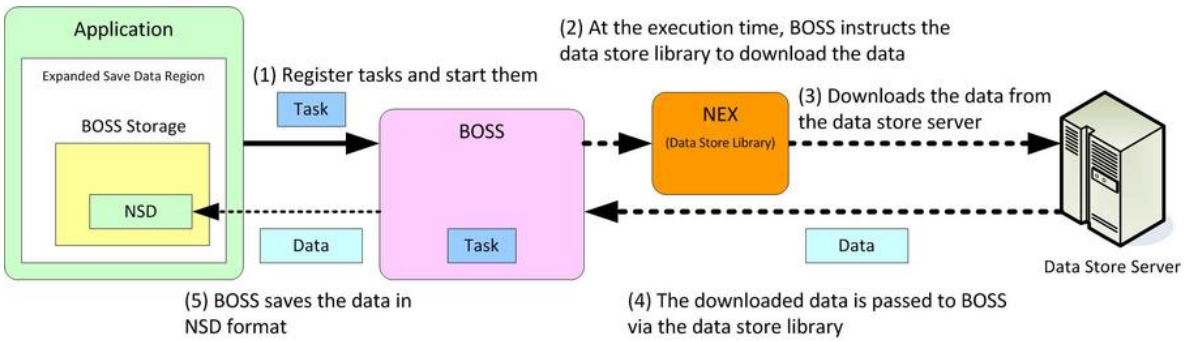
## 4.2.6. Data Store Download Tasks

Data Store download tasks are tasks that download data using the NEX library's data store features.

Because information for the data that has been downloaded is recorded, only that data viewable by the user which has not yet been downloaded is included in the download.

A Data Store download task can use the NEX library Data Store features, even when the application is not running (such as when it is in Sleep Mode) by communicating via SpotPass.

**Figure 4-12. Downloading With a Data Store Download Task**


Application
Expanded Save Data Region
BOSS Storage
NSD
(1) Register tasks and start them
Task
Data
BOSS
Task
(2) At the execution time, BOSS instructs the data store library to download the data
NEX
(Data Store Library)
(3) Downloads the data from the data store server
Data
Data Store Server
(5) BOSS saves the data in NSD format
(4) The downloaded data is passed to BOSS via the data store library

Data that has been downloaded is saved in BOSS storage in NSD format. For this reason, the same function is used for loading data as was used for the data that was downloaded using the NADL task. When BOSS storage is full, data is automatically deleted starting with the earliest data (data with earlier serial numbers). This process is also the same as is used on data downloaded using an NADL task.

By performing notification issue settings when a task is registered, a notification can be posted to the notifications list when new data is downloaded.

> **Note:** This feature uses the NEX library, but you do not need to include the NEX library in the application.
>
> For more information about data store features, see the NEX Library documentation.

# 4.3. Presence Feature

With Nintendo DS NITRO-DWC and TWL-DWC, friend-to-friend communications had to be managed by separate implementation into each application. With 3DS, these features are supported by the SDK.

> **Note:** For more information about the presence features, see the *System Application and Applet Specification*.

## 4.3.1. Overview

With 3DS, systems have a shared friend list available to all games from which users can add and remove other users from the HOME Menu by standard system functions. The user does not need to independently implement an interface for management of the friend list by each application.

Also, the user does not need to add other users separately for each application. Other functionality for getting and displaying information such as friends' Mii characters, status messages, and the title of the game currently being played is now built into the system. Applications can get or set some part of this data by calling the appropriate API function, and information about oneself and one's friends is automatically synchronized. Because this communication process is executed by daemons running in the background, applications do not need to keep track of detailed information for sending and receiving.

The Join-In feature expands on network functionality. This feature supports inviting friends to join your group while an online game is in progress or to join a friend's group by virtue of linkage with the NEX library. For more information about using the Join-In function, see the *NEX Programming Manual: Server Services*.

## 4.3.2. User Accounts

The presence feature includes individual accounts. Information in those accounts is shared with friends. The account ID is automatically assigned by the system.

### 4.3.2.1. Types of User IDs

Two types of IDs are used to identify a user of the presence feature: the *principal ID* and the *local friend code*. Both IDs are assigned different values each time the system is initialized.

#### Principal ID

A completely unique ID assigned upon first connection to the friend server. Although the ID will not collide with other IDs, the value is invalid and cannot be used until you connect to the friend server for the first time.

#### Local friend code

This user ID is used to identify users for the presence features. It is generated based on the system's unique ID. Although extremely rare, collisions cannot be entirely avoided.

There are also two secondary IDs, *friend keys* and *friend codes*, that are generated based on these IDs.

#### Friend key

The concept of this ID is to handle both the principal ID and local friend code. As a rule, this

ID uses the local friend code until a principal ID is issued, and then it uses the principal ID from then on.

**Friend code**

This 12-digit number is generated from the principal ID that is meant to be user-friendly in its format. It is generated based on the principal ID. Because it has a one-to-one correspondence with principal IDs, the two codes can be mutually converted. It includes simple error detection capability.

## 4.3.2.2. Account Information

The following information can be obtained from friends or made available to friends using the presence feature.

- Friend key
- The title of the Currently Playing game
- Game mode description string
- Join-in information for the Currently Playing game (for linking with the NEX library)
- Mii
- Screen name (Mii name)
- System profile information (country and region settings)
- Friendships
- Favorite title
- Status message

Applications can set or get some part of the information in this table. In addition, some items that cannot be directly set or obtained by applications can have their setting configured or checked by the other system features, such as the HOME Menu.

**Table 4-24. Setting and Getting Presence Information**

| Item | Setting/Editing Local Information | | Getting/Checking Friend Information | |
|---|---|---|---|---|
| | Application | System Features | Application | System Features |
| Friend code | No | No | No | Yes |
| Friend key | No | No | Yes | No |
| The title of the Currently Playing game | No | Yes | No | Yes |
| Game mode description string | Yes | No | No | Yes |
| Join-in information about the Currently | | | | |

| | | | | |
|---|---|---|---|---|
| Playing game | Yes | No | Yes | No |
| Mii | No | Yes | Yes | Yes |
| Screen name (Mii name) | No | Yes | Yes | Yes |
| System profile information (Country and region settings) | No | Yes | Yes | No |
| Friendships | No | Yes | Yes | Yes |
| Favorite title | No | Yes | No | Yes |
| Status message | No | Yes | No | Yes |

## 4.3.3. Managing the Friend List

With 3DS, you can add users to and remove users from the friend list by using the Friend List system applet and then saving it on the system.

This friend list can be used in common with all 3DS game titles.

### 4.3.3.1. Registering Friends on the Friend List

There are three methods of registering another user on the friend list. These methods are described below.

#### Entering a Friend Code

You can register another party by entering their friend code on the Friend List system applet. When entered, the validity of the friend code is verified by connecting to the friend server. The environment must allow connection to the Internet. The user can check the user's friend code on the Friend List system applet.

The side entering the friend code for the other party enters a wait state to establish friendship with that other party until the other party adds you to their friend list. Friendship is established after both parties have confirmed registration of each other.

#### Registering Using Local Communication

You can register another party in your friend list using local communication by selecting **Local** on the Friend List system applet. Connection to the Internet is not required in this case.

The friendship is established without waiting under this method, because parties are added to each other's friend lists immediately.

#### Registering Within an Application

You can also register another party in your friend list within an application by sending and receiving the required registration information.

When performing registration this way, confirmation must be obtained to ensure that both parties consent to friend registration before proceeding.

**Note:** For more information about registration within an application, see 4.3.9. Friend Registration. The UI associated with registration must be implemented in the application.

**Warning:** Friend registration within an application that is performed via StreetPass or local communication, where it is not possible to identify the other party in advance, is prohibited.

For more information, see the Friend Registration section in *3DS Guidelines: Internet Communication*.

## 4.3.3.2. Friendship Status

Friendships are established when two parties register each other on their friend lists. Friendships transition through the following states, depending on their respective registration status.

### Waiting to Establish Friendship

In this state, one party has added another to his or her friend list and is waiting to be added to the other party's friend list. When the other party adds the first party to the friend list, that fact is registered on the friend server. After the other party adds the first party to his or her friend list, the state changes to friendship established the next time a connection is made to the friend server.

### Friendship Established

In this state, both parties have registered the other as a friend. Friendships must be in this status to share information such as connection status and the game title that is currently being played in real-time.

**Note:** The friend server maintains a list of users that are registered in each user's friend list. Two users who are registered on each other's friend list are treated as being in the friendship established state. As such, friendships that have been established using local communication are not treated as being in the friendship established state until both users have gone online at least once after establishing the friendship. Usually,

there is no reason to recognize this status, but it is called local friend status to distinguish it.

#### Friendship Canceled

This is the state, for example, when you have been removed from the friend list of another party with whom you had established a friendship. Although the other party is not just removed unexpectedly from your own friend list, after this state results, the other party appears as if offline, even if they are actually online. Follow-up access is still possible for information received from the other party before the friendship was canceled. Friendship established status can be restored if the other party registers your friend code again.

## 4.3.4. Initialization and Finalization

You must initialize the library by first calling `nn::friends::Initialize` to use API functions in the FRIENDS library. Also, call `nn::friends::Finalize` to finalize the library when it is not going to be used anymore.

The FRIENDS library has a counter for keeping the initialization count internally. If `nn::friends::Initialize` has been called multiple times, you must call `nn::friends::Finalize` the same number of times to completely shut down the library. Use `nn::friends::IsInitialized` to check whether the library has already been initialized.

**Code 4-53. API Functions for Initializing and Finalizing**

```
nn::Result nn::friends::Initialize();
nn::Result nn::friends::Finalize();
bool nn::friends::IsInitialized();
```

## 4.3.5. Online and Offline

The term *online* refers to the status when 3DS is connected to the friend server, while *offline* is used to describe the status when not connected. In online mode, not only can you share information with your friends, such as the title you are currently playing and the game mode, you can also receive the same information from friends who are also online. In offline mode, though you can get some information such as your friends' names, their Mii characters, and profile information, there is no guarantee that the information is the latest available.

### 4.3.5.1. Autonomous Connection and Login

There are two methods of putting the 3DS in online mode.

## Autonomous Connection

Even without an explicit request from the application, the friend presence daemon autonomously attempts to connect to the friend server and enter online mode if possible. This behavior is managed by the daemon manager and may stop without confirmation, as determined by the daemon manager.

In addition, if you want the application to explicitly stop this behavior, you must instruct the daemon manager to stop execution of the friend presence daemon.

## Login

If the application is actively using the presence feature, you can issue a login request to enter online mode regardless of autonomous connection status.

Use `nn::friends::Login` to issue a login request. If this function succeeds, a connection process runs in the background, and the `nn::os::Event` object passed in `pEvent` at the time the process exits enters signal status. Results of asynchronous processing can be obtained using `nn::friends::GetLastResponseResult`. If a login request is issued and the system is already online, `nn::friends::HasLoggedIn` returns `true`.

**Note:** Do not, however, issue a login request for titles that do not support online play. To issue a login request, you must have already established an Internet connection using the AC (automatic connection) library. For this reason, be sure that login requests are performed only when applications also request an Internet connection.

Even if login is not required, you can cancel a login request by logging out. Again, even without logging out explicitly, login requests are canceled when the FRIENDS library is closed or when an application exits. If autonomous connection is still enabled even after logout, online status will be maintained.

**Code 4-54. API Function for Login**

```
nn::Result nn::friends::Login(nn::os::Event* pEvent);
nn::Result nn::friends::Logout();
nn::Result nn::friends::GetLastResponseResult();
bool nn::friends::HasLoggedIn();
```

**Code 4-55. Sample Code for Login Process**

```
nn::Result result;
nn::os::Event event(false);

result = nn::friends::Login(&event);
if (result.IsSuccess())
{
    // Generates an asynchronous process if Login succeeds.
    if (event.Wait(nn::fnd::TimeSpan::FromMinutes(2)))
    {
        result = nn::friends::GetLastResponseResult();
```

```
            if (result.IsSuccess())
            {
                // Login succeeded.
            }
            else
            {
                // Error handling.
            }
        }
        else
        {
            // Request must be canceled because failure of the login process has
not been established.
            nn::friends::Logout();
        }
    }
    else
    {
        // Error handling.
    }
    event.Finalize();
```

**Warning:** Asynchronous processes started by `nn::friends::Login` processes terminate immediately if online status has been established by autonomous connection. The system takes several seconds to several dozen seconds to transition to online status by asynchronous process. If the user cancels the login request without waiting for this asynchronous process to end, explicitly call `nn::friends::Logout` to cancel the login request or the process of connecting to the friend server connection will continue to run in the background.

## 4.3.5.2. Synchronizing Information With the Friend Server and Friends

User information is sent to the friend server while online and then shared with friends via the friend server.

Although local information can be updated even offline, it is not sent immediately to the friend server. Updating local information offline is synchronized by sending the information to the friend server when the system goes online and connects to the friend server.

On the other hand, updates conducted while online are immediately sent to the friend server by a friend presence daemon running in the background. However, because the library restricts the send interval to the friend server to once per 10 seconds at most for any single item, even if information is updated at a shorter interval than this, immediate synchronization with the friend server and friend is impossible. Do not communicate with the friend server at a very high frequency. For more information, see the *CTR Guidelines*.

**Code 4-56. API Functions for Updating Local Information**

```
nn::Result nn::friends::UpdateGameModeDescription(
        const char16 description[nn::friends::MODE_DESCRIPTION_SIZE]);
```

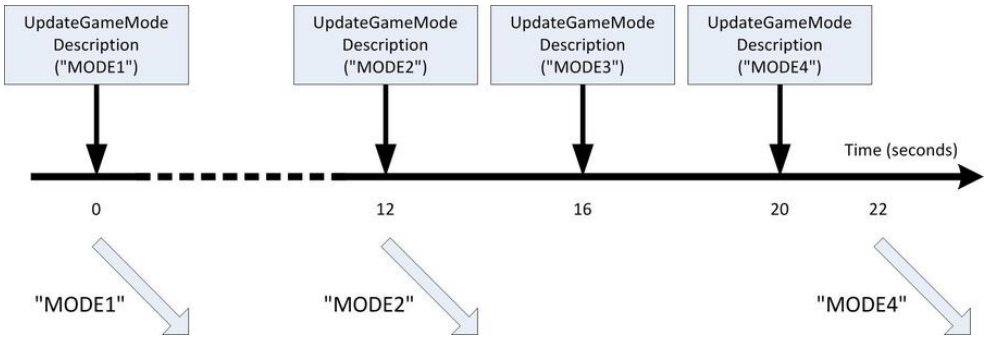
The `nn::friends::UpdateGameModeDescription()` function updates the text string describing your own game mode.

The string specified by *`description`* is applied to what is displayed on your own friend card. Up to two lines of text having a length equal to 16 characters (in Japan, European, U.S. regions: "%"; in other regions: Japanese hiragana characters, Chinese kanji characters, Hangul characters, and so on) of the internal font at maximum width can be displayed on the friend card. The characters that can be used for this string differ depending on the system region, as shown in the following table. Characters that cannot be displayed are converted to question marks (0xE011).

**Table 4-25. Characters Displayable on Friend Cards**

| Region | Characters That Can Be Displayed |
|---|---|
| Japan, US, Europe | Characters included in the internal font Japanese and Western characters |
| China | Characters included in the internal font simplified Chinese characters |
| South Korea | Characters included in the internal font Hangul characters |
| Taiwan | Characters included in the internal font traditional Chinese characters |

**Figure 4-13. Example of Background Communication When the Text String Describing the Local Game Mode Is Updated at a Short Interval**




Some user information is saved on the friend server. Even if your friend is not online at the same time you are, this allows information in effect the last time your friend was online to be obtained from the friend server when you go online. Because the friend information obtained most recently is saved by the 3DS, you can even get friend information when you are offline. However, the friend information obtained when offline may not be the latest information. Information that is meaningless unless both you and your friend are online at the same time, such as the game mode being played, is not saved on the friend server and cannot be obtained unless both you and your friend are online.

- Information that can be obtained only when both parties are online:
  - The title of the Currently Playing game
  - Game mode description string
  - Join-in information about the Currently Playing game
- Information that can be obtained at any time:
  - Friend key

- Mii
- Screen name (Mii name)
- System profile information (country and region settings)
- Friendships
- Favorite title
- Status message

For more information about the API functions used to get friend information, see 4.3.6. Getting Friend Information.

## 4.3.5.3. Private Mode

Local information can be set as private from the Friend List system applet.

There are two disclosure levels.

- Only the Currently Playing game is kept private.
- All presence information is kept private.

There are two disclosure levels.

If you keep all presence information private, you can make it appear to friends that you are offline even though you are online. In other words, information such as your Currently Playing game is not sent to your friends, and your connection status is handled as if you are offline. In addition, because even the friend server will not update your information, settings made the last time you were online with public settings will be used by the friend server for items such as your Mii and system profile information. Friend information can be obtained in the same way as when presence information has been made public.

## 4.3.5.4. Getting Local System Information

Use the following functions to get the local system's own information.

**Code 4-57. Functions for Getting the Local System's Information**

```
nn::friends::PrincipalId nn::friends::GetMyPrincipalId();
bool nn::friends::IsMyPreferenceValid();
nn::Result nn::friends::GetMyPreference(
        bool* pIsPublicMode, bool* pIsShowGameName);
nn::Result nn::friends::GetMyProfile(nn::friends::Profile* pProfile);
nn::Result nn::friends::GetMyPresence(nn::friends::MyPresence* pMyPresence);
nn::Result nn::friends::GetMyScreenName(char16 screenName[SCREEN_NAME_SIZE]);
nn::Result nn::friends::GetMyMii(nn::mii::StoreData* pMiiData);
```

If you have never connected to the friend server, `INVALID_PRINCIPAL_ID` is returned as the principal ID.

If you have never started the friend list (a system applet), the `IsMyPreferenceValid()` function returns `false`, because you have never touched the friend notification settings. In addition, the default value (`true`) is stored for the various arguments of the `GetMyPreference()` function.

If you have not created a Personal Mii and you have never started the Friend List system applet, the system profile information obtained by the `GetMyProfile()` function reflects either the settings of the system at initial setup, or the current System Settings.

If you have not used Mii Maker to create a Personal Mii, the `GetMyScreenName()` function returns an empty string starting with NULL (meaning that the second and subsequent characters are undefined). The `GetMyMii()` function returns empty Mii data. You must use the CFL library to determine whether the obtained Mii data is valid and to display the data.

## 4.3.6. Getting Friend Information

When getting friend information using the FRIENDS library, specify the friend using the friend's friend key or principal ID. You can get friend keys for users registered in your friend list using `nn::friends::GetFriendKeyList`, or you can specify friend keys and principal IDs obtained externally via other communication.

### 4.3.6.1. Getting a Friend Key

To get the friend key of a user registered on the friend list, use `nn::friends::GetFriendKeyList`. This function attempts to get the friend keys for the number of users specified by *`size`* in order, beginning from the *`offset`* friend in the friend list, and stores them in the buffer pointed to by *`pFriendKeyList`*. It returns the number of friend keys actually stored in a buffer pointed to by *`pNum`*. If the number of friend keys obtained is less than that specified by *`size`*, the values in the buffer for friend keys that could not be stored cannot be guaranteed.

**Code 4-58. Function for Getting Friend Keys**

```
nn::Result nn::friends::GetFriendKeyList(
        nn::friends::FriendKey* pFriendKeyList, size_t* pNum,
        size_t offset = 0, size_t size = nn::friends::FRIEND_LIST_SIZE);
```

The local friend codes obtained through local communications are scrambled to protect privacy, so they cannot be used as is to specify friends. The FRIENDS library has prepared an API for unscrambling these local friend codes.

**Code 4-59. Function for Unscrambling Local Friend Codes**

```
nn::Result nn::friends::UnscrambleLocalFriendCode(
        nn::friends::LocalFriendCode* pLocalFriendCodeList,
        const nn::uds::ScrambledLocalFriendCode* pScrambledLocalFriendCodeList,
        size_t size = 1);
```

For *pScrambledLocalFriendCodeList* and *size*, specify the list of (scrambled) local friend codes that was obtained through local communications and the number of entries in the list.

The result after unscrambling is stored in *pLocalFriendCodeList*, so set this parameter to a value large enough to hold the number of entries specified by *size*. Only the codes for friends can be unscrambled. The local friend codes for users that have not been registered to the friend list are treated as INVALID_LOCAL_FRIEND_CODE.

When implementing your application, note the possibility that friends may have the same local friend code. The unscrambled local friend codes are not unique values, so in extremely rare instances the friend list may contain multiple friends with the same local friend code.

## 4.3.6.2. Getting Friend Information

Pass a list of friend keys or principal IDs to the API function used to get the friend information. As a result, friend information is stored in a buffer passed to the first argument of each function in the same number and order as the list of friend keys or principal IDs passed to the function. If the user corresponding to the specified friend key or principal ID is not included in the friend list, invalid data is returned.

**Note:** There is no API for getting friend information by specifying a list of local friend codes. Although it is possible to store a local friend code in a friend key and pass that, collisions, while extremely rare, cannot be completely avoided due to the nature of local friend codes. More than one user having the same local friend code could be registered in the friend list. As long as the friend key has been obtained from the friend list, the user in the list can be uniquely identified even if there is more than one user having the same friend code in the list, because there is never more than one user in the list without a principal ID (when there is a collision prior to establishing a friendship, the last-registered user is the one who remains).

**Code 4-60. Functions for Getting Friend Information**

```
nn::Result nn::friends::GetFriendPresence(
        nn::friends::FriendPresence* pFriendPresenceList,
        const nn::friends::FriendKey* pFriendKeyList, size_t size = 1);
nn::Result nn::friends::GetFriendPresence(
        nn::friends::FriendPresence* pFriendPresenceList,
        const nn::friends::PrincipalId* pPrincipalIdList, size_t size = 1);
nn::Result nn::friends::GetFriendScreenName(
        char16 (*pScreenNameList)[nn::friends::SCREEN_NAME_SIZE],
        const nn::friends::FriendKey* pFriendKeyList, size_t size = 1,
```

```
        bool replaceForeignCharacters = true, u8* pFontRegionList = NULL);
nn::Result nn::friends::GetFriendScreenName(
        char16 (*pScreenNameList)[nn::friends::SCREEN_NAME_SIZE],
        const nn::friends::PrincipalId* pPrincipalIdList, size_t size = 1,
        bool replaceForeignCharacters = true, u8* pFontRegionList = NULL);
nn::Result nn::friends::GetFriendMii(
        nn::mii::StoreData* pMiiDataList,
        const nn::friends::FriendKey* pFriendKeyList, size_t size = 1);
nn::Result nn::friends::GetFriendMii(
        nn::mii::StoreData* pMiiDataList,
        const nn::friends::PrincipalId* pPrincipalIdList, size_t size = 1);
nn::Result nn::friends::GetFriendProfile(
        nn::friends::Profile* pProfileList,
        const nn::friends::FriendKey* pFriendKeyList, size_t size = 1);
nn::Result nn::friends::GetFriendProfile(
        nn::friends::Profile* pProfileList,
        const nn::friends::PrincipalId* pPrincipalIdList, size_t size = 1);
nn::Result nn::friends::GetFriendAttributeFlags(
        bit32* pAttributeFlagsList,
        const nn::friends::FriendKey* pFriendKeyList, size_t size = 1);
nn::Result nn::friends::GetFriendAttributeFlags(
        bit32* pAttributeFlagsList,
        const nn::friends::PrincipalId* pPrincipalIdList, size_t size = 1);
```

## 4.3.7. Notifications

To find changes in connection status with the friend server and updates to friend information without polling, you can get change notifications from the friend presence daemon by specifying the items required by the application. (Note that these notifications are separate from the user-facing messages delivered to the Notifications built-in application.)

### 4.3.7.1. Types of Notifications

Applications can receive the following nine types of notifications. In addition to the types of changes, the friend key of the friend is also notified if notification is related to friends. When a notification arrives, friend information previously obtained by a function call may have become out-of-date. Be sure to refresh this information when convenient for the application.

**Table 4-26. Types of Notifications**

| Definition | Description (including functions that must be called again to refresh the data) |
|---|---|
| NOTIFICATION_ONLINE | Indicates that the local system came online. An invalid value is stored in the friend key. |
| NOTIFICATION_OFFLINE | Indicates that the local system went offline. An invalid value is stored in the friend key. |
| NOTIFICATION_FRIEND_ONLINE | Indicates that a friend has come online.<br>GetFriendPresence, GetFriendScreenName, GetFriendMii, GetFriendProfile, GetFriendAttributeFlags |

| | |
|---|---|
| NOTIFICATION_FRIEND_PRESENCE | Indicates that the friend's game mode has changed.<br>GetFriendPresence |
| NOTIFICATION_FRIEND_MII | Indicates that the friend's Mii has changed.<br>GetFriendScreenName, GetFriendMii |
| NOTIFICATION_FRIEND_PROFILE | Indicates that the friend's profile has changed.<br>GetFriendProfile |
| NOTIFICATION_FRIEND_OFFLINE | Indicates that a friend has gone offline.<br>GetFriendPresence |
| NOTIFICATION_BECOME_FRIEND | Indicates that a friendship has been established with a user.<br>GetFriendPresence, GetFriendScreenName, GetFriendMii, GetFriendProfile, GetFriendAttributeFlags |
| NOTIFICATION_INVITATION | Indicates that the local system has received an invitation from a friend.<br>GetFriendPresence |

## 4.3.7.2. Notification-Related Functions

Use the following functions to get notifications from the friend presence daemon.

**Code 4-61. Notification Functions**

```
nn::Result nn::friends::AttachToEventNotification(nn::os::Event* pEvent);
nn::Result nn::friends::SetNotificationMask(bit32 mask);
u32 nn::friends::GetEventNotification(
        nn::friends::EventNotification* pEventNotificationList,
        size_t size = 1, bool* pHasOverflowed = NULL);
```

### Event Registration

If you call `nn::friends::AttachToEventNotification()` and pass a pointer to an initialized `nn::os::Event` object in the *`pEvent`* parameter, that event is signaled when a notification occurs.

### Setting a Notification Mask

To specify the types of notifications you want to receive, pass a bitwise OR of your desired values of the `nn::friends::NotificationMask` enumerated type in the *`mask`* parameter of the `nn::friends::SetNotificationMask()` function. If the daemon sends notifications of types not enabled by this mask, the `nn::os::Event` object passed to `nn::friends::AttachToEventNotification` is not signaled and these notifications are excluded from the notification history you can get with the `nn::friends::GetEventNotification()` function.

### Getting Notification Details

Get notification details using the `nn::friends::GetEventNotification()` function. The return value is the number of notifications obtained by the call. Call this function until it returns zero to get all of the notifications accumulated in the daemon's buffer in the same order the notifications occurred.

> **Note:** Notifications start accumulating when the FRIENDS library is initialized using `nn::friends::Initialize`. If you want notifications to start accumulating from zero (that is, to start accumulating only notifications that are newer than the current time), you can exclude all notifications by passing `0` in the *`mask`* parameter of the `nn::friends::SetNotificationMask()` function and then delete all notifications received in the past by calling the `nn::friends::GetEventNotification()` function once.

> **Warning:** The notification history (the daemon's buffer) stores up to 128 previous notifications, including private notifications sent to the system. If this buffer overflows, the earliest notification in the list is deleted first. If this buffer does overflow and earlier notifications are discarded, your application will no longer be able to track what portions of the friend presence information are out of date. In this case, the application must immediately call the functions to get the local system's connection status and friend information again.

## 4.3.8. Error Handling

If a function in the FRIENDS library fails when called to get the error code, call the `nn::friends::ResultToErrorCode()` function, passing the `nn::Result` object as the *`result`* parameter.

#### Code 4-62. Function for Getting Error Codes

```
u32 ResultToErrorCode(const nn::Result& result);
```

If this function returns zero, there is no need to display the error code. There is also no need to display the error code if the user can continue playing without experiencing any problems, such as when login fails but the application will immediately retry automatically.

> **Warning:** When converting the content of the `nn::Result` object into an error code, the `nn::friends::ResultToErrorCode()` function sometimes appends information related to current daemon status. Consequently, when getting an error code, call

`nn::friends::ResultToErrorCode` immediately after receiving a failed result.

## 4.3.9. Friend Registration

Friend registration within an application can be performed by mutually receiving the information for registration that the `nn::friends::GetMyApproachContext()` function gets, after which both parties pass the received information to the `nn::friends::AddFriendWithApproach()` function.

**Code 4-63. Functions Used for Friend Registration Within an Application**

```
nn::Result nn::friends::GetMyApproachContext(
                        nn::friends::ApproachContext* pApproachContext);
nn::Result nn::friends::AddFriendWithApproach(
                        nn::os::Event* pEvent,
                        const nn::friends::ApproachContext& approachContext);
```

To use these functions, the FRIENDS library must be initialized. It is not necessary to log in to a friend's server. In addition, the CFG Library must also be initialized in advance using the `nn::cfg::Initialize()` function.

The `nn::friends::ApproachContext` structure obtained by the `GetMyApproachContext()` function is already encrypted, so it is not necessary to encrypt the data for sending and receiving on the application side.

Friend registration completes when calling the `AddFriendWithApproach()` function is successful (for example, when the result returned for the `IsSuccess()` function is `true`). Asynchronous processing then occurs. Asynchronous processing does not occur for any other `Result`. When asynchronous processing completes, the event is passed to *pEvent* and the application is notified of the signaled state. You can get the results of asynchronous processing with the `nn::friends::GetLastResponseResult()` function. Note that library specifications do not support simultaneous handling of multiple instances of asynchronous processing.

Friends registered with this function are local friends, and you cannot get presence information until both parties connect to the Internet and the friend relationship can be established on the friend server. Asynchronous processing merely tests the registration process on the friend server. Even when registration succeeds with both users online, presence information may not be immediately available after asynchronous processing completes.

To display information for friend registration that has been received in the application (the `ApproachContext` structure), you can get the display name and Mii data using the following functions.

**Code 4-64. Functions for Getting Information for Friend Registration**

```
nn::Result nn::friends::GetApproachContextScreenName(
                        char16 screenName[nn::friends::SCREEN_NAME_SIZE],
                        const nn::friends::ApproachContext& approachContext,
                        bool replaceForeignCharacters, u8* pFontRegion);
```

```
nn::Result nn::friends::GetApproachContextMii(
                          nn::mii::StoreData* pMiiData,
                          const nn::friends::ApproachContext& approachContext);
```

The `GetApproachContextScreenName()` function returns the friend display name passed to *`screenName`* to the buffer. If `true` is specified for `replaceForeignCharacters` at this point, characters that cannot be displayed in the system region standard font (which are included in the display name acquired) are replaced by question marks and acquired in that form. To display these characters in another region without replacing them with question marks, specify `false` and then pass the pointer to the u8 type variable to *`pFontRegion`*. The region with the standard font that can display the characters is returned to *`pFontRegion`* by the value set in the `nn::friends::FontRegion` enumerator.

The `GetApproachContextMii()` function returns the information required for displaying the friend's Mii to the buffer passed to *`pMiiData`*. The separately available Face library is also required in order to handle the Mii information acquired.

CONFIDENTIAL

# 5. Supplemental Communication Libraries

The following libraries have been provided to assist applications using wireless communication.

- Blocked-User List
- Profanity List
- Notifications
- Accounts

This chapter describes the programming procedures and information required to develop an application that uses these libraries.

## 5.1. Blocked-User List

The blocked-user list is a feature provided to prevent the wide spread of inappropriate user-generated content (UGC) over peer-to-peer (P2P) communication.

### 5.1.1. Overview

If UGC is being managed by a particular server, the administrator checks all instances of UGC that have been registered and deletes inappropriate UGC. However, because content is directly transmitted without going through a server when using P2P communication, there is no way to handle inappropriate UGC.

The blocked-user list is intended to prevent the spread of inappropriate UGC as much as possible, even when P2P communication is used.

Authors of inappropriate UGC are registered in a list on the system (the local blocked-user list) by the blocked-user list feature. Inappropriate UGC is restricted by preventing UGC from authors registered in the list from being displayed.

Users can perform the following operations on the local blocked-user list on the HOME Menu.

- Delete all registered UGC author information (lift the blocked-user list restrictions)

The blocked-user list is only used to register and access the IDs of UGC authors. Even if the system receives UGC that was created by an author on the blocked-user list, the content will not be deleted automatically. If you receive UGC that must comply with the blocked-user list, you must check whether the author ID is on the blocked-user list from within your application, and take appropriate measures (such as deleting the content). The application is also responsible for adding UGC author IDs to the blocked-user list.

**Note:** The local blocked-user list is saved to the shared expanded save data region.

## 5.1.2. Initialization and Finalization

Use the UBL library to use the blocked-user list feature in an application. Initialize and finalize the UBL library using the `nn::ubl::Initialize()` and `nn::ubl::Finalize()` functions.

**Code 5-1. Initializing and Finalizing the UBL Library**

```
void nn::ubl::Initialize(void);
void nn::ubl::Finalize(void);
```

## 5.1.3. Local Blocked-User List

Each system saves a single local blocked-user list that lists authors of received UGC. Users know this list as the **Blocked-User Settings**.

The UGC author information of UGC that the user deems to be inappropriate is registered on the local blocked-user list. Call `nn::ubl::Entry` to register entries on the local blocked-user list.

**Code 5-2. Registering Entries on the Local Blocked-User List**

```
nn::Result nn::ubl::Entry(u64 id, nn::fnd::DateTime *dt);
```

Specify the UGC author ID to register on the local blocked-user list in *id*. The function returns an error if the local author (the user of the local system) tries to register himself or herself.

The UGC author ID represents the ID of the author who created the UGC. This ID is used only for the blocked-user list. Each system has a unique 8-byte ID. Although it cannot be used to identify a particular system, the same ID is used even when UGC is created with a different application. If the application allows received UGC to be edited, the ID of the user doing the editing (the editor ID) is registered for edited UGC data rather than the ID of the user who first created that UGC (the original author ID).

**Note:** For information about what the application must support, see the Blocked-User List section of the guidelines.

Specify the registration date and time in *dt*. The date and time that the UGC author ID was registered on the blocked-user list as UGC author information is used to determine which information to delete if the local blocked-user list becomes full.

Up to 1000 instances of UGC author information can be registered. If the number of registered entries reaches the maximum value at time of registration, entries are automatically overwritten beginning from the earliest.

## 5.1.4. Blocked-User List Check

When receiving or displaying UGC targeted by the blocked-user list (see Blocked-User List in the guidelines), make sure to check whether the UGC author ID of that UGC is registered on the local blocked-user list.

You can use `nn::ubl::IsExist` to check the UGC author ID.

**Code 5-3. Checking the Blocked-User List**

```
bool nn::ubl::IsExist(u64 authorId, u32 titleId, u64 dataId);
```

Specify the UGC author ID to check in *authorId*. If the received UGC can be edited, make sure to check both the original author ID and editor ID.

The arguments *titleId* and *dataId* will be used with an extended feature in the future. Currently, their use is unsupported and they are ignored if specified.

Make sure that UGC on the blocked-user list is either deleted by the application or not displayed by the application.

## 5.1.5. UGC Browse Mode

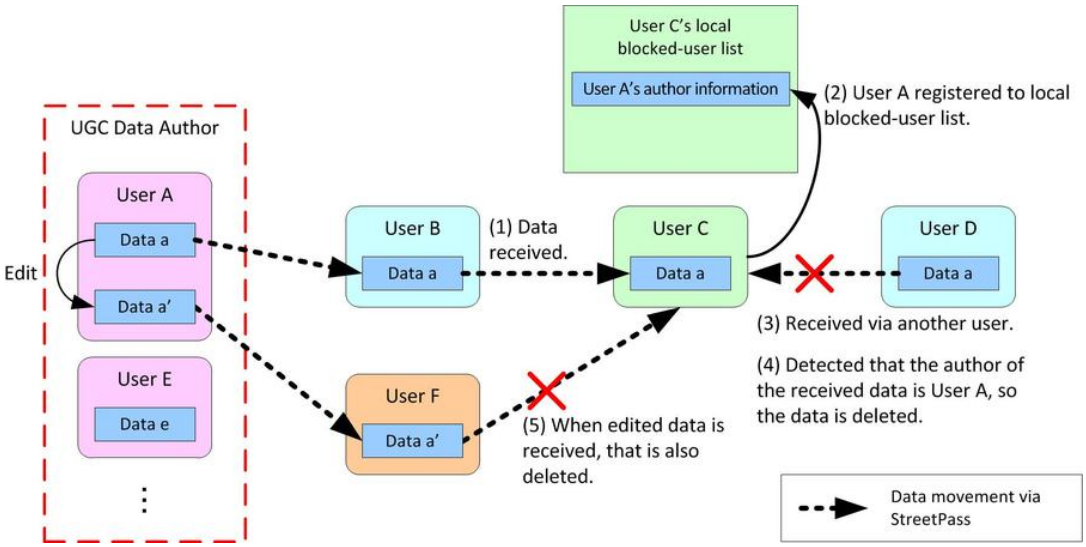
UGC browse mode is a mode (browsing window) that allows you to browse UGC content that may be displayed within an application. Applications that allow the creation of UGC must include a UGC browse mode.

In UGC browse mode, users can display the UGC of authors not yet registered on the local blocked-user list within the associated application. Implement this feature so that users can register browsed UGC that they determine to be inappropriate on their local blocked-user list.

## 5.1.6. Local Blocked-User List Registration Flow

The following figure represents an example of receiving data via StreetPass. It shows the execution flow when registering an author on the local blocked-user list.

**Figure 5-1. Local Blocked-User List Registration Flow**




1. Assume that data including UGC created by User A (**Data a**) is received by User C from User B.
2. User C looks at **Data a** inside an application and, deeming it inappropriate, selects **Data a** from UGC browse mode. The application then displays a message and checks whether the user is registered in the local blocked-user list. User A is registered in the local blocked-user list when **Yes** is selected.
   "Place content author on blocked-user list? Yes. No."
3. Next, assume that the same **Data a** is received from User D.
4. When using StreetPass, the system detects that User A, the creator of **Data a**, is registered in the local blocked-user list and deletes **Data a** the next time User C opens the corresponding application after **Data a** has been received. (See the Blocked-User List section in the guidelines.)
5. After this, even if User A edits **Data a** to create **Data a'**, and this **Data a'** is then received from User F, **Data a'** will be deleted because the author ID of **Data a'** is registered in the local

blocked-user list.

## 5.2. Profanity List

Whenever UGC that contains text is received, check against the profanity list to determine whether the text contains any profanity. Run this check not only for content received via the Internet, but also when content is received via local communications such as StreetPass.

Run this check not only for content received via the Internet, but also when content is received via local communications such as StreetPass. When systems with different language settings are communicating, the profanity check may be run on only the sender or only the receiver or it may be run on both sides.

Nintendo provides the NGC library for the CTR-SDK to check for profanity. Unlike the DWC library prepared for the Wii system, this NGC library can check for profanity without the need for going through a server.

### 5.2.1. Initialization and Finalization

Use the `nn::ngc::ProfanityFilter` class to check for profanity. You must initialize the FS library before using this class.

**Code 5-4. Initializing the nn::ngc::ProfanityFilter Class**

```
class nn::ngc::ProfanityFilter
{
    ProfanityFilter();
    ProfanityFilter(const nn::WithInitialize&);
    ProfanityFilter(uptr pWorkMemory);

    nn::Result Initialize();
    nn::Result Initialize(uptr pWorkMemory);
}
```

The working memory used for the profanity check must be allocated either by the library from a memory block, or by the application. The size required for working memory is `nn::ngc::ProfanityFilter::WORKMEMORY_SIZE` (65,536 bytes).

If an instance is generated in an argument by the `nn::WithInitialize()` function, or if an instance is generated by a constructor without an argument, and the `Initialize()` function is called without an argument, the working memory is allocated from a memory block. In either case, note that the use of memory blocks (the `nn::os::MemoryBlock` class) must be enabled beforehand.

If an instance is generated by specifying the starting address of the working memory in

*pWorkMemory*, or if an instance is generated by a constructor without an argument and the `Initialize()` function is called with an argument, the working memory must be allocated by the application. For this working memory, allocate a contiguous region that is least `nn::ngc::ProfanityFilter::WORKMEMORY_SIZE` (65,536 bytes) in size and has `nn::ngc::ProfanityFilter::WORKMEMORY_ALIGNMENT` (4 bytes) alignment.

After the check is completed and the instance is no longer needed, call the `Finalize()` function to finalize the process.

**Code 5-5. Finalizing the nn::ngc::ProfanityFilter Class**

```
class nn::ngc::ProfanityFilter
{
    nn::Result Finalize();
}
```

After finalization you can release the working memory. If the working memory was allocated from a memory block, it is released during the finalization process.

## 5.2.2. Checking for Profanity

The profanity check for illegal strings is conducted by the `ProfanityFilter` class `CheckProfanityWords` member function.

**Code 5-6. Profanity Check**

```
class nn::ngc::ProfanityFilter
{
    nn::Result CheckProfanityWords(
        bit32* pCheckResults, const wchar_t** ppWords, size_t nWordCount);
    nn::Result CheckProfanityWords(
        bit32* pCheckResults, nn::ngc::ProfanityFilterPatternList nPatternCode,
        const wchar_t** ppWords, size_t nWordCount);
    nn::Result CheckProfanityWords(
        bit32* pCheckResults, bool bCommunicateWithOtherRegions,
        const wchar_t** ppWords, size_t nWordCount);
}
```

Each overload is checked from the top in all regions, and all languages and pattern lists are specified and checked (for region and language combinations). Pattern lists are automatically determined and checked based on the appropriate regions of the system.

The arguments *ppWords* and *nWordCount* specify the string array to check and the number of strings in that array. Specify a string terminated with a UTF16-LE NULL terminator. Multiple strings can be checked at the same time, and this is faster than checking them one at a time.

The results of the profanity determinations are stored in *pCheckResults*. Specify a `bit32`-type array of *nWordCount* or more elements. Each determination is returned as a bit flag. If a bit flag is up (the bit is 1), the string has been judged to be profane based on one of the pattern lists. The bit in the pattern list corresponds to the value that is left-shifted by 1 bit in

`nn::ngc::ProfanityPatternList`. If no problems are found in any of the pattern lists, `0` is stored in *`pCheckResults`*.

The pattern lists that are used for profanity checks are specified in *`nPatternCode`* using the values defined in `nn::ngc::ProfanityPatternList`. The following pattern lists are defined in `nn::ngc::ProfanityPatternList`.

**Table 5-1. Pattern Lists**

| Definition | Region | Language |
|---|---|---|
| PATTERNLIST_JAPAN_JAPANESE | Japan | Japanese |
| PATTERNLIST_AMERICA_ENGLISH | Americas | American English |
| PATTERNLIST_AMERICA_FRENCH | Americas | French (Canadian) |
| PATTERNLIST_AMERICA_SPANISH | Americas | Spanish (Latin America) |
| PATTERNLIST_EUROPE_ENGLISH | Europe | British English |
| PATTERNLIST_EUROPE_FRENCH | Europe | French |
| PATTERNLIST_EUROPE_GERMAN | Europe | German |
| PATTERNLIST_EUROPE_ITALIAN | Europe | Italian |
| PATTERNLIST_EUROPE_SPANISH | Europe | Spanish |
| PATTERNLIST_EUROPE_DUTCH | Europe | Dutch |
| PATTERNLIST_KOREA_KOREAN | South Korea | Korean |
| PATTERNLIST_CHINA_SIMP_CHINESE | China | Chinese (Simplified) |
| PATTERNLIST_EUROPE_PORTUGUESE | Europe | Portuguese |
| PATTERNLIST_EUROPE_RUSSIAN | Europe | Russian |
| PATTERNLIST_AMERICA_PORTUGUESE | Americas | Portuguese (Brazil) |
| PATTERNLIST_TAIWAN_TRAD_CHINESE | Taiwan | Chinese (Traditional) |
| PATTERNLIST_TAIWAN_ENGLISH | Taiwan | English (Taiwan) |

In the overload with the *`bCommunicateWithOtherRegions`* parameter, the system region determines which pattern lists must be used for the profanity check. For this reason, you must initialize the CFG library if this overload is being called. If `false` is specified for this parameter, the profanity check is conducted using only the pattern lists that match the system region. This parameter is not currently referenced, and the profanity check is conducted as if `false` were specified.

The profanity check can sometimes block processing and take time to complete, so we recommend creating a separate thread for these checks.

## 5.2.3. Checking Numbers

The checking performed by the `nn::ngc::ProfanityFilter` class determines whether a string

includes the @ symbol, which may be used in the notation for an email address, but the checking cannot determine whether the string contains many numerals, such as might be the case for a telephone number.

To check for numerals contained in strings, call the `nn::ngc::CountNumbers()` function. You do not need to initialize the `nn::ngc::ProfanityFilter` class to call this function

**Code 5-7. Checking Numbers Contained in Strings**

```
int nn::ngc::CountNumbers(const wchar_t *pString);
```

The *`pString`* parameter specifies the string to check. Specify a string terminated with a UTF16-LE NULL terminator.

As the return value, the function returns the number of numerals in the string. If an error occurs, the function returns a negative value.

## 5.2.4. Text Check

The `nn::ngc::ProfanityFilter` class `MaskProfanityWordsInText()` function checks long strings (such as sentences) for profanity, and when a string thought to be problematic is found, it can be replaced with asterisks. For a profanity check on shorter strings, such as display names, use the `CheckProfanityWords()` function.

**Code 5-8. Checking for Profanity in Text**

```
class nn::ngc::ProfanityFilter
{
    nn::Result MaskProfanityWordsInText(
        int* pProfanityWordCount, wchar_t* pText);
    nn::Result MaskProfanityWordsInText(
        int* pProfanityWordCount,
        nn::ngc::ProfanityFilterPatternList nPatternCode, wchar_t* pText);
    nn::Result MaskProfanityWordsInText(
        int* pProfanityWordCount,
        bool bCommunicateWithOtherRegions, wchar_t* pText);
    void SetMaskMode(bool bOverWrite);
}
```

Each overload is checked from the top in all regions, and all languages and pattern lists are specified and checked (for region and language combinations). Pattern lists are automatically determined and checked based on the appropriate regions of the system.

The pointer to the text targeted for a check is passed to *`pText`*. When a pointer to an `int`-type variable that is not NULL is passed to *`pProfanityWordCount`*, the number of times the problem string appears in the text is returned.

For more information about specifying pattern lists using *`nPatternCode`* and specifying *`bCommunicateWithOtherRegions`* see the `CheckProfanityWords()` function.

The profanity check can sometimes block processing and take time to complete, so we recommend creating a separate thread for these checks.

When problem strings are replaced with asterisks by the `MaskProfanityWordsInText()` function at *`bOverWrite`* with the `SetMaskMode()` function, it also specifies whether to replace the string with the same number of characters (`true`: default) or with only one character (`false`). Note that when the string is replaced by the same number of characters, the new string displayed in a proportional font may extend beyond the original frame due to differences in character width between the original characters and the asterisks.

# 5.3. Notifications

Notifications is a feature of the HOME Menu that works like the Wii Message Board to show notifications from Nintendo and from applications. Notifications can be posted from the application.

## 5.3.1. Initialization and Finalization

To post notifications from the application you must use the NEWS library. Before posting notifications, use the `nn::news::Initialize()` function to initialize the NEWS library. After you no longer need to post notifications, call the `nn::news::Finalize()` function.

**Code 5-9. Initializing and Finalizing the NEWS Library**

```
nn::Result nn::news::Initialize();
nn::Result nn::news::Finalize();
```

## 5.3.2. Posting Notifications

To post a notification, call the `nn::news::PostNews()` function.

**Code 5-10. Posting Notifications**

```
nn::Result nn::news::PostNews(
                    const wchar_t* subject, const wchar_t* message,
                    const u8* picture = NULL, size_t pictureSize = 0,
                    u32 serialId = 0, u32 dataVersion = 0, u64 jumpParam = 0);
```

You must specify both the title (*`subject`*) and the body (*`message`*) for this post. Both strings must be terminated with a UTF16-LE NULL terminator. Use the `0x000A` (LF) control character for line

breaks.

When an image is attached to the notification, the *picture* and *pictureSize* parameters specify the image data and the image size.

The *serialId* and *dataVersion* parameters specify the notification's serial ID and data version number. At present, specify `0`.

The *jumpParam* argument specifies the parameters when jumping from the notification list to the application. The application can call `nn::news::IsFromNewsList` to detect when the notification list is activated, and can get the parameters specified at the *jumpParam* argument.

The title, body, and image attachment specified by these parameters are copied to inside the function, so they can be released after posting has completed.

The following table provides specifications for the title, body, and image attachment. For more information, such as the restriction on the interval between postings, see the *CTR Guidelines*.

**Table 5-2. Notification Specifications**

| Item | Specification |
|---|---|
| Title | The character encoding is UTF16-LE. The total number of characters including the string terminator must be no greater than `nn::news::SUBJECT_LEN`.<br>The length of characters that are displayed is equal to the length of 17 characters of the system fonts at maximum width (in Japan, European, U.S. regions: "%"; in other regions: Japanese hiragana characters, Chinese kanji characters, Hangul characters, and so on). If the maximum display length is exceeded, the character size is reduced by up to 80%. If it still exceeds the maximum display length, the excess portion is hidden. |
| Body | The character encoding is UTF16-LE. The total number of characters including the string terminator must be no greater than `nn::news::MESSAGE_LEN`.<br>A number of characters equal in length to 18 characters of the internal font at maximum width is displayed in a single line. |
| Attached images | MPO format. 3D pictures are also possible. The size in bytes must be no greater than `nn::news::PICTURE_SIZE`. |

The following errors are returned if the characters specified for title or body exceed the specifications, or if the specified image for attachment is in an invalid format or exceeds the data size limit.

**Table 5-3. Errors Returned When Posting Notification**

| Return Value | Cause of Error |
|---|---|
| `ResultInvalidSubjectSize` | The specified title has too many characters. |
| `ResultInvalidMessageSize` | The specified body has too many characters. |
| `ResultInvalidPictureSize` | The image attachment exceeds the limit size. |
| `ResultInvalidPicture` | The image data is in an unsupported format. |

## 5.3.3. Posting Notifications With URLs

A notification with a URL is a normal notification with a URL provided. When a URL is provided along with a notification and displayed in the notification list, a button is also displayed that allows you to open your Internet browser to the page with the specified URL.

**Code 5-11. Functions Used With Notifications With URLs**

```
nn::Result nn::news::PostNewsUrl(
                    const wchar_t* subject, const wchar_t* message,
                    const u8* url, u8* workBuf,
                    const u8* picture = NULL, size_t pictureSize = 0,
                    u32 serialId = 0, u32 dataVersion = 0, u64 jumpParam = 0);
size_t nn::news::GetWorkBufferSizeForNewsUrl(
                    const wchar_t* message, const u8* url);
```

You can specify parameters other than *url* and *workBuf* in the same way as for the nn:news::PostNews() function.

For the *url* parameter, specify the URL to be provided in UTF-8 encoding. The maximum URL length that can be provided including the NULL terminator is nn::news::MESSAGE_URL_URL_SIZE bytes. Because the URL name uses the same region where the main text is stored, use the following formula to calculate the number of characters allowed when a URL is provided.

> nn::news::MESSAGE_LEN - (Number of characters in the URL, including the NULL terminator) / 2

The size of the allocated work buffer returned by the GetWorkBufferSizeForNewsUrl() function is specified in *workBuf*. If the main text and URL total a number of characters exceeding the allowable length, or if an illegal argument such as an illegal pointer is passed, 0 is returned.

# 5.4. Accounts

The account library (the ACT library) enables applications to use the 3DS platform's account system features. For an overall picture of the account system, see the *CTR Account System Developer's Guide* included in the CTR-SDK.

By using the ACT library, applications can access the Nintendo Network account information that is cached in the CTR system and perform authentication processes based on the registered information.

## 5.4.1. Initialization and Finalization

Use the nn::act::Initialize() function to initialize the ACT library.

If you are using ACT library functions that involve Internet communications, such as for the network

clock or the login applet, complete a connection to the Internet with the ACT library before you call those functions. When you initialize the ACT library, you must specify a buffer for communications for these functions that perform Internet communications.

> **Note:** For the ACT library functions that are currently public functions, you do not need to specify a buffer for communications.

The buffer you pass for communications **must be allocated from outside of device memory**. The starting address of the buffer must be aligned to `nn::act::BUFFER_ALIGNMENT` (4,096 bytes), and the size of the buffer must be a multiple of `nn::act::BUFFER_UNITSIZE` (4,096 bytes).

The `nn::act::Initialize()` function can be called multiple times. However, for the communications buffer, the specified buffer is used when the ACT library has not been initialized, and it is not used when it has been initialized.

The number of calls to the `nn::act::Initialize()` function is counted and recorded. Until you call the `nn::act::Finalize()` function as many times as the `nn::act::Initialize()` function, the ACT library is considered to be initialized.

## 5.4.2. Nintendo Network Account Information

The application can use the ACT library to use Nintendo Network account information.

Because most of the functions that get Nintendo Network account information return locally cached information, these functions do not perform communications when they execute.

The table shows the correspondence between library functions and the Nintendo Network information they get. For more information, see the API reference.

**Table 5-4. Functions for Getting the Different Nintendo Network Account Information**

| Information | Function | Comments |
|---|---|---|
| Network account | `IsNetworkAccount` | Determines whether the system account is a network account. |
| Account ID | `GetAccountId` | |
| Principal ID | `GetPrincipalId` | The principal ID obtained by the ACT library is the ID that was issued by the account server when the Nintendo Network account was registered.<br>Although the name is the same, the principal ID obtained by the FRIENDS library is the ID issued by the friend server and differs from the one obtained by the ACT library. Be careful not to confuse these two IDs, and make mistakes like passing the principal ID obtained by the ACT library to the functions of the FRIENDS library. |
| Password | — | Cannot be obtained. |
| Email | | Contact Nintendo if your plans require getting this |

| | | |
|---|---|---|
| address | — | information. |
| Date of birth | — | Contact Nintendo if your plans require getting this information. |
| Age | `IsOverAge` | Can only get information as to whether a user meets a specified age. For more information, see 5.4.2.1. Age Determination. |
| Country of residence | `GetCountry` | Returned in the ISO 3166-1 alpha-2 format. |
| Region of residence | `GetSimpleAddressId`<br>`GetWiiUSimpleAddressId` | |
| Time zone | `GetTimeZoneId` | Returns the time zone of the region of residence in the tz database format. |
| Time offset | `GetUtcOffset` | Returns the difference between UTC time and the time in the region of residence, in seconds.<br>This process takes daylight savings time into account, but depending on when the information was cached, the time difference might not be accurate. |
| Gender | — | Contact Nintendo if your plans require getting this information. |

### 5.4.2.1. Age Determination

The `nn::act::IsOverAge()` function determines whether the Nintendo Network account user is at least the age specified in the *age* parameter.

Age is determined using the birth date registered in the Nintendo Network account and the network clock. If the CTR system account is not a Nintendo Network account, or if the network clock is not enabled, this function always returns `false`. For more information about the network clock, see 5.4.3. Network Clock.

## 5.4.3. Network Clock

If the network clock is not enabled and the time is not corrected before you call functions that use the network clock, the processes of those functions will be based on the wrong time. Use the `nn::act:: IsNetworkTimeValidated()` function to check whether the network clock is available, and if it is not, call the `nn::act::InquireNetworkTime()` function.

Note that the `nn::act::InquireNetworkTime()` function communicates with the account server and blocks execution of the application until its process is completed.

The network clock becomes available for use after this function communicating with the account server has ended execution normally.

## 5.4.4. Login Applet

The login applet communicates with the account server, working on behalf of the application to authenticate the account and get various service tokens and the like. Control does not return to the application until its series of processes have completed.

The login applet authenticates the account, and returns an error if the system account is not a Nintendo Network account. We recommend that you check the account by calling the `nn::act::IsNetworkAccount()` function beforehand and that you not use the login applet if it is not a Nintendo Network account.

**Warning:** The login applet does not check the Parental Controls settings. If your application has features that can be restricted by the Parental Controls, make sure that the application appropriately reflects the setting. For example, if the Parental Controls have been set to restrict all services that are available after account authentication, do not even bother calling the login applet.

**Note:** Like other library applets, the login applet supports preloading. You also must support and check to determine whether control returned to the applet due to the pressing of the HOME Button or the POWER Button or a software reset immediately after control was returned to the application.

### 5.4.4.1. Service Tokens for Independent Services

For services that use independent servers, you can use the `nn::act::applet::AcquireIndependentServiceToken()` function to get a service token that verifies that the connected user has received Nintendo account authentication.

**Code 5-12. Function for Getting the Service Token for Independent Service**

```
nn::Result nn::act::applet::AcquireIndependentServiceToken(
    char*           pServiceToken,
    const char*     pClientId,
    u32             reusableRangeInSeconds = 0
);
```

If a nonzero value is set in the *`reusableRangeInSeconds`* parameter, the cached service token is returned without communicating with the account server, providing that a certain amount of time (specified as a number of seconds) has not passed since the last time a service token was obtained from the account server. If the service token is invalid, for example because time has expired or due to a denial by the independent server, specify `0` for this parameter to ensure that you can get another service token.

The login applet performs communications internally. You do not need to specify a buffer for communications when you initialize the ACT library.

**Code 5-13. Example of Error Handling When Getting a Service Token for Independent Service**

```
char serviceToken[NN_ACT_INDEPENDENT_SERVICE_TOKEN_SIZE];
nn::Result result = nn::act::applet::AcquireIndependentServiceToken(
    serviceToken, CLIENT_ID_INDEPENDENT, SERVICE_TOKEN_REUSABLE_RANGE );

if ( result.IsSuccess() )
{
    // Stores the service token in serviceToken.
}
else if ( nn::act::ResultCanceled::Includes( result ) )
{
    // Explicitly canceled by user, so go to the finalization sequence.
}
else if ( nn::act::ResultApplicationUpdateRequired::Includes( result ) )
{
    // An application update is required, so jump to the Nintendo eShop patch
page (optional).
}
else
{
    // Some error other than a cancellation has occurred, so show the error
code.
    u32 networkErrorCode = nn::act::GetErrorCode( result );

    // Initialize the configuration structure.
    nn::applet::AppletWakeupState wstate;
    nn::erreula::Parameter ere_param;    // Error EULA configuration structure
    ere_param.config.errorType = nn::erreula::ERROR_TYPE_ERROR_CODE;
    ere_param.config.errorCode = networkErrorCode;
    ere_param.config.upperScreenFlag = nn::erreula::UPPER_SCREEN_NORMAL;
    ere_param.config.homeButton = true;
    ere_param.config.softwareReset = false;
    ere_param.config.appJump = false;

    nn::erreula::StartErrEulaApplet( &wstate, &ere_param );
}
```

### 5.4.4.2. Service Tokens for Miiverse

Use the `nn::act::applet::AcquireOlvServiceToken()` function to get Miiverse service tokens. Service tokens for Miiverse are needed when using the OLV library.

The login applet performs communications internally. You do not need to specify a buffer for communications when you initialize the ACT library.

The errors that require handling are the same as those for 5.4.4.1. Service Tokens for Independent Services.

## 5.4.5. Creating UUIDs

Use the `nn::act::GenerateUuid()` function to create UUIDs (Universally Unique Identifiers) that can be allocated to automatically generated objects and the like.

The created UUIDs comply with RFC 4122 - Version 1 specifications. The format might be changed in the future, but only in ways that will not compromise uniqueness of the UUIDs. For this reason, use the values returned by the `nn::act::GenerateUuid()` function as the UUIDs without making any changes to them. Do not make use of only portions of these UUIDs, and do not extract time information or other information from these UUIDs for other uses.



# 6. Debugging Libraries

**Warning:** No developer support is offered for these libraries, and these libraries generally cannot be included in retail products. Contact Nintendo (support@noa.com) if you want to do so.

Limited debugging libraries are available for checking network connectivity, such as socket communication using the wireless module.

**Table 6-1. Debugging Libraries**

| Feature | Library Name | Namespace | Description |
|---|---|---|---|
| Socket Communication | SOCKET | `nn::socket` | Library for implementing data transmission to and from a remote host via sockets. |
| SSL Communication | SSL | `nn::ssl` | Helper library for secure communication via SSL. |
| HTTP Communication | HTTP | `nn::http` | Library for communication via HTTP. |

**Note:** The HTTP library conducts socket communications internally, and the process is not affected by the socket API operations of the application.

For example, the application can call functions like `nn::socket::Initialize()` and `nn::socket::Finalize()` while using the HTTP library without affecting the processing of the HTTP library.

## 6.1. Socket Communication

The SOCKET library for socket communication handles communication with a remote host via sockets bound to addresses and port numbers. Sockets provide a means for low-level communication between the local machine and remote hosts for transmitting data, waiting for connections from remote hosts, and connecting from the local host to a remote host.

## 6.1.1. Initializing

Call the `nn::socket::Initialize()` function to initialize the SOCKET library.

**Code 6-1. Initializing the SOCKET Library**

```
size_t nn::socket::GetRequiredMemorySize(size_t bufferSizeForSockets,
                                         s32 maxSessions);
nn::Result nn::socket::Initialize(uptr bufferAddress, size_t bufferSize,
                                  s32 bufferSizeForSockets, s32 maxSessions);
```

The *bufferAddress* and *bufferSize* parameters specify the starting address of the working memory used by the library and the size of that memory. The starting address of the working memory must be `nn::socket::BUFFER_ALIGNMENT` (4,096 bytes) aligned and **must be allocated from non-device memory**. Call the `nn::socket::GetRequiredMemorySize()` function, passing the size of the send/receive buffer to allocate for an entire socket (*bufferSizeForSockets*) and the maximum number of sessions (*maxSessions*) as arguments, to get the required size of the working memory. The size of the communication buffer (*bufferSizeForSockets*) specified here must be a multiple of `nn::socket::BUFFER_UNITSIZE_FOR_SOCKETS` (4,096 bytes).

You can use the `nn::socket::SetSockOpt()` function to specify the send/receive buffer to assign to each socket. By default, a 16-KB send/receive buffer (8 KB for sending and 8 KB for receiving) is assigned to a single TCP socket, and a 32-KB send/receive buffer is assigned to a single UDP socket. Even if the maximum number of sockets is 1, allocate at least 64 KB with *bufferSizeForSockets*.

The *maxSessions* argument specifies the number of threads that use sockets (the maximum number of sessions). The library can handle calls to blocking library functions from the specified number of threads. Strictly speaking, the library can handle function calls from more than the specified number of threads (as long as they are not calling blocking library functions), but we do not recommend this approach. Both asynchronous and synchronous library function calls block, until there is a free session, if they exceed the maximum number of sessions. Processing blocks until there is an open session, even when other conditions to unblock have been satisfied, such as when data has finished being received or sent. Note that this may cause processing that is meant to be asynchronous to work synchronously instead. You can check socket states with a single call to the `nn::socket::Poll()` function, which allows you to reduce the number of required sessions.

**Warning:**The size of the send/receive buffer assigned by default may be subject to change.

# 6.1.2. Creating a Socket

Both the remote host and the local host must create sockets before socket communication can occur. Call the `nn::socket::Socket()` function to create a socket. The number of sockets that an application can use simultaneously is limited to 16.

> **Note:** The number of sockets that can be used simultaneously may be subject to change.

**Code 6-2. Creating a Socket**

```
s32 nn::socket::Socket(s32 af, s32 type, s32 protocol);
```

*af* may only specify PF_INET.

*type* specifies the type of socket to create as either SOCK_STREAM for stream sockets or SOCK_DGRAM for datagram sockets. Stream sockets require that the sockets on both sides of the communication establish a connection, whereas datagram sockets allow for one-way sending or receiving of data blocks. Stream sockets also guarantee the order of arrival of data blocks. Datagram sockets do not, but as a consequence transmit faster.

*protocol* specifies the protocol the socket uses, but currently only `0` may be specified. A *protocol* value of 0 indicates the use of the default protocol for the protocol family and type specified in the *af* and *type* parameters. The default protocol is TCP for stream sockets and UDP for datagram sockets.

If the return value is 1 or greater, the return value is a socket descriptor for identifying the socket. A value of 0 or less indicates an error. The following table lists the errors that may occur.

**Table 6-2. Errors When Creating a Socket**

| Return Value | Error Description |
|---|---|
| ENETRESET | Library not initialized. |
| EAFNOSUPPORT | Specified protocol family not supported. |
| EPROTONOSUPPORT | Specified protocol is not supported. |
| EMFILE | No more socket descriptors can be created. |
| ENOMEM | Insufficient memory to allocate working memory. |
| EPROTOTYPE | Specified socket type not supported. |
| EINVAL | Invalid call. |
| ENETDOWN | Local network interface is down. |
| EBUSY | The internal process is busy. There may be a shortage of sessions specified by `Initialize()`. |

## 6.1.3. Binding Addresses and Port Numbers

A socket cannot be used for communication unless it is bound to a socket address indicating which address and which port number to use. A freshly created socket does not have a socket address bound to it. Call the `nn::socket::Bind()` function to bind a socket address to a socket.

**Code 6-3. Binding an Address and Port Number**

```
s32 nn::socket::Bind(s32 s, const nn::socket::SockAddrIn* sockAddr);
```

*s* specifies the socket descriptor for the created socket. An error occurs if the specified socket already has a socket address bound to it.

*sockAddr* specifies a pointer to the `nn::socket::SockAddrIn` structure in which the socket address (address family, port number, address) has been set. Only IPv4 addresses are supported.

A return value of 0 indicates success. A nonzero return value indicates an error. The following table lists the errors that may occur.

**Table 6-3. Errors When Binding Addresses and Port Numbers**

| Return Value | Error Description |
|---|---|
| ENETRESET | Library not initialized. |
| EBADF | The specified socket descriptor is not valid. |
| ENETDOWN | Local network interface is down. |
| EINVAL | Invalid operation (such as when the socket is already bound). |
| EAFNOSUPPORT | Specified address family not supported. |
| EADDRINUSE | Specified socket address already in use. |
| EBUSY | The internal process is busy. There may be a shortage of sessions specified by `Initialize()`. |

## 6.1.4. Operating Modes

Call the `nn::socket::Fcntl()` function to get or set a socket's operating mode.

**Code 6-4. Getting and Setting a Socket's Operating Mode**

```
s32 nn::socket::Fcntl(s32 s, s32 cmd, s32 val);
```

*s* specifies the socket descriptor for the socket for which to set or get the operating mode.

*cmd* specifies whether to set (F_SETFL) or get (F_GETFL) the socket's operating mode.

*val* specifies either 0 or a bitwise OR of the flags to set for the operating mode. This parameter is

ignored when the *cmd* value is F_GETFL.

The only flag that may be set is O_NONBLOCK for non-blocking operation. Sockets that do not have this flag set operate in blocking mode instead. Sockets are set to blocking mode when first created.

When *cmd* is F_GETFL, the return value is a bitwise OR of the operating mode flags that have been set. When *cmd* is F_SETFL, a return value of 0 indicates successful completion. For any *cmd* value, a negative return value indicates an error. The following table lists the errors that may occur.

**Table 6-4. Errors When Getting or Setting a Socket's Operating Mode**

| Return Value | Error Description |
|---|---|
| ENETRESET | Library not initialized. |
| EBADF | The specified socket descriptor is not valid. |
| ENETDOWN | Local network interface is down. |
| EINVAL | Invalid process (invalid value specified for *cmd*). |
| EBUSY | The internal process is busy. There may be a shortage of sessions specified by `Initialize()`. |

## 6.1.5. Waiting for Connections

Datagram sockets can send and receive data right after a socket address is bound, but stream sockets must establish a connection before data may be transmitted.

The stream socket on the server side creates a queue for connection requests from client-side stream sockets by calling the `nn::socket::Listen()` function, and then accepts incoming connection requests by calling the `nn::socket::Accept()` function.

**Code 6-5. Waiting for Connections**

```
s32 nn::socket::Listen(s32 s, s32 backlog);
s32 nn::socket::Accept(s32 s, nn::socket::SockAddrIn* sockAddr);
```

The *s* parameter for both functions specifies the socket to use for incoming requests. This socket must have been created as a stream socket, and must have a socket address bound.

*backlog* specifies the maximum number of items in the queue used as the socket listening backlog. A value of 0 or a negative number is treated as a value of 1.

*sockAddr* specifies a pointer to an `nn::socket::SockAddrIn` structure storing the socket address of the socket that received the request.

A return value of 0 for the `nn::socket::Listen()` function indicates success. A negative return value indicates an error.

For the `nn::socket::Accept()` function, if there is no socket awaiting connection in the queue, processing blocks if the socket used to wait for connections is in synchronous mode. A return value

of 1 or greater is the socket descriptor for a newly created socket that has the same address as the socket used to wait for connections. A value of 0 or less indicates an error.

The following table lists the errors that may occur.

**Table 6-5. Errors When Waiting for Connections**

| Return Value | Error Description |
|---|---|
| ENETRESET | Library not initialized. |
| EBADF | The specified socket descriptor is not valid. |
| EOPNOTSUPP | Unsupported operation. |
| EINVAL | Invalid operation. |
| ENOBUFS | Insufficient resources. |
| EAGAIN | No socket is waiting for connections. (Non-blocking mode) |
| ECONNABORTED | Connection attempt canceled. |
| EMFILE | No more socket descriptors can be created. |
| ENOMEM | Insufficient memory to create the listening backlog. |
| EWOULDBLOCK | Same as EAGAIN. |
| ENETDOWN | Local network interface is down. |
| EBUSY | The internal process is busy. There may be a shortage of sessions specified by `Initialize()`. |

## 6.1.6. Connecting to a Remote Host

The client-side stream socket attempts a connection to the server-side stream socket using the `nn::socket::Connect()` function, and cannot send or receive data until a connection is established. A call to `nn::socket::Connect` by a datagram socket only overwrites the sending target socket address.

**Code 6-6. Connecting to a Remote Host**

```
s32 nn::socket::Connect(s32 s, const nn::socket::SockAddrIn* sockAddr);
```

*s* specifies the socket descriptor for the socket used for the connection. If this socket has no socket address bound, the function binds the socket to an unused local socket address.

*sockAddr* specifies a pointer to an `nn::socket::SockAddrIn` structure to which the connection target socket address has been set.

A return value of 0 indicates success. A nonzero return value indicates an error. The following table lists the errors that may occur.

**Table 6-6. Errors When Connecting to a Remote Host**

| Return Value | Error Description |
|---|---|
| ENETRESET | Library not initialized. |
| EAFNOSUPPORT | Specified protocol family not supported. |
| EBADF | The specified socket descriptor is not valid. |
| EALREADY | Already attempting to connect in non-blocking mode. |
| ECONNREFUSED | Connection reset by remote host after the local and remote host simultaneously became open. |
| ECONNRESET | Connection was reset. |
| EINPROGRESS | Specified socket is currently changing state. |
| EINVAL | Invalid process (invalid value specified for *sockAddr*). |
| EISCONN | Socket already in use and cannot be used for other processing. |
| ENETDOWN | Local network interface is down. |
| ENETUNREACH | Could not reach the connection target. |
| ENOBUFS | Failed to allocate the temporary socket address for the empty socket. |
| ETIMEDOUT | Response from connection target timed out. |
| EBUSY | The internal process is busy. There may be a shortage of sessions specified by `Initialize()`. |

If the socket specified by *s* is a datagram socket, the function only overwrites the socket's sending target socket address with the socket address set in *sockAddr*, with no difference in operation for different operating modes.

If the socket specified by *s* is a stream socket and the operating mode is synchronous, the function blocks until a connection is established. If the operating mode is asynchronous, the function returns immediately, and the `nn::socket::Poll()` function can be used to check if a connection has been established.

**Code 6-7. Checking a Connection to a Remote Host**

```
struct nn::socket::PollFd
{
    s32         fd;
    s32         events;
    s32         revents;
};
s32 nn::socket::Poll(nn::socket::PollFd fds[], u32 nfds, s32 timeout);
```

The `nn::socket::Poll()` function surveys multiple socket descriptors for sockets that can send and receive.

*fds* specifies an array of `nn::socket::PollFd` structures that contain the socket descriptors and survey conditions. The structure member variable `fd` contains a socket descriptor to survey, and the `events` member variable contains the survey conditions. The `revents` member stores the survey results. Set it to `0` when calling the function.

If a connection could not be established by a call to `nn::socket::Connect`, the `POLLRDNORM` and `POLLWRNORM` flags are set in `revents`, and an error is then returned when data is transmitted.

The following table lists the flags set for the survey conditions set in `events` and for the survey results set in `revents`.

**Table 6-7. Flags Set for Survey Conditions and Survey Results**

| Flag | Description |
|---|---|
| `POLLRDNORM` | Indicates an ability to receive. |
| `POLLRDBAND` | Indicates an ability to receive (for priority data). |
| `POLLPRI` | Indicates an ability to receive (for highest-priority data). |
| `POLLWRNORM` | Indicates an ability to send. |
| `POLLWRBAND` | Indicates an ability to send (for priority data). |
| `POLLERR` | Indicates that an error occurred. Only set for survey results. |
| `POLLHUP` | Indicates that the socket was disconnected. Only set for survey results. |
| `POLLNVAL` | Indicates that an invalid socket descriptor was specified. Only set for survey results. |
| `POLLIN` | Indicates an ability to receive. Bitwise OR of `POLLRDNORM` and `POLLRDBAND`. |
| `POLLOUT` | Indicates an ability to send. Same as `POLLWRNORM`. |

*`nfds`* specifies the number of elements in the array specified in *`fds`*.

*`timeout`* specifies the number of milliseconds to wait until timing out when no sockets are found matching the conditions. Specify as a number greater than 0 or as `INFTIM` (no timeout).

A call to `nn::socket::Poll` blocks until a socket is found that matches the conditions. However, the block is released if the specified timeout period passes with no matching sockets found, if the socket is disconnected, or if an error occurs in the socket.

A return value of 1 or greater is the number of sockets that matched the conditions. A return value of 0 indicates that the function timed out. A negative return value indicates an error. The following table lists the errors that may occur.

**Table 6-8. Errors When Checking Connection to a Remote Host**

| Return Value | Error Description |
|---|---|
| `ENETRESET` | Library not initialized. |
| `ENETDOWN` | Local network interface is down. |
| `EINVAL` | Invalid operation. |
| `EBUSY` | The internal process is busy. There may be a shortage of sessions specified by `Initialize()`. |

## 6.1.7. Receiving Data

You can receive data using the `nn::socket::RecvFrom`, `nn::socket::Recv`, or

nn::socket::Read() functions. For stream sockets, data cannot be received unless a connection has been established on both the server and client side.

**Code 6-8. Receiving Data**

```
s32 nn::socket::RecvFrom(s32 s, void* buf, s32 len, s32 flags,
                         nn::socket::SockAddrIn* sockFrom);
s32 nn::socket::Recv(s32 s, void* buf, s32 len, s32 flags);
s32 nn::socket::Read(s32 s, void* buf, s32 len);
s32 nn::socket::SockAtMark(s32 s);
```

*s* specifies a socket descriptor for a local socket.

*buf* specifies a pointer to a buffer for storing the received data, and *len* specifies the size of that buffer in bytes.

*flags* specifies flags for any special operations to carry out when receiving. If MSG_DONTWAIT is specified, the function does not block even if the socket is in synchronous mode. If MSG_PEEK is specified, data that is received but the state is not changed, and the same data can be received again. If MSG_OOB is specified, out-of-band data can be received. Check whether the nn::socket::SockAtMark() function's return value is 1 to determine whether out-of-band data is included in the receivable data. This function can also be used to check whether the last byte of TCP protocol "urgent data" is at the start of receivable data.

If the *sockFrom* parameter specifies a pointer to an nn::socket::SockAddrIn structure, the socket address of the sender of the data is stored.

If receiving via a stream socket and the operating mode is synchronous, the function blocks until data is received. If the mode is asynchronous or if *flags* specifies MSG_DONTWAIT, only that data that is receivable when the function is called is stored in the buffer, and the function does not block.

If receiving via a datagram socket, all of the data to receive must be received in a single function call. If the data exceeds the size of the receive buffer and MSG_PEEK is not specified for *flags*, any data that does not fit in the buffer is discarded. The maximum amount of data that can be received by a single function call is 1500 bytes.

A return value of 1 or greater is the number of bytes of received data. A return value of 0 when using a stream socket indicates that the remote host has finished sending data. The return value will never be 0 when using a datagram socket. A negative return value indicates an error. The following table lists the errors that may occur.

**Table 6-9. Errors When Receiving Data**

| Return Value | Error Description |
|---|---|
| ENETRESET | Library not initialized. |
| EBADF | The specified socket descriptor is not valid. |
| EAGAIN | No socket is waiting for connections (The non-blocking mode or *flags* specifies MSG_DONTWAIT). |
| EINVAL | Invalid operation. |
| EOPNOTSUPP | Unsupported operation. |

| | |
|---|---|
| ENOTCONN | Specified socket is not connected to the remote host. |
| ECONNRESET | Connection was reset. |
| EINTR | Process interrupted. |
| ETIMEDOUT | The process timed out. |
| ENETDOWN | Local network interface is down. |
| EWOULDBLOCK | Same as EAGAIN. |
| EBUSY | The internal process is busy. There may be a shortage of sessions specified by Initialize(). |

## 6.1.8. Sending Data

You can receive data using the nn::socket::SendTo, nn::socket::Send, or nn::socket::Write() functions. For stream sockets, data cannot be sent unless a connection has been established on both the server and client side.

**Code 6-9. Sending Data**

```
s32 nn::socket::SendTo(s32 s, const void* buf, s32 len, s32 flags,
                       const nn::socket::SockAddrIn* sockTo);
s32 nn::socket::Send(s32 s, const void* buf, s32 len, s32 flags);
s32 nn::socket::Write(s32 s, const void* buf, s32 len);
```

*s* specifies a socket descriptor for a local socket.

*buf* specifies a pointer to a buffer for storing the data to send, and the *len* parameter specifies the size of that buffer in bytes.

*flags* specifies flags for any special operations to carry out when sending.

If *sockTo* specifies a pointer to an nn::socket::SockAddrIn structure, the function sends the data to the socket address set in the structure.

The function blocks until data sending has completed.

If sending with a datagram socket, the maximum amount of data that can be sent by a single function call is 1500 bytes.

A return value of 0 indicates success. A negative return value indicates an error. The following table lists the errors that may occur.

**Table 6-10. Errors When Sending Data**

| Return Value | Error Description |
|---|---|
| ENETRESET | Library not initialized. |
| EAFNOSUPPORT | Specified protocol family not supported. |
| EBADF | The specified socket descriptor is not valid. |
| | If in blocking mode, the prior send attempt was blocked. |

| | |
|---|---|
| EAGAIN | If the socket is a stream socket in non-blocking mode, data to be sent out-of-band cannot be reserved for sending in the internal send buffer. |
| ECONNRESET | Connection was reset. |
| EINTR | Process interrupted. |
| EINVAL | Invalid operation. |
| EOPNOTSUPP | Unsupported operation. |
| EDESTADDRREQ | If the socket is a stream socket, the socket is being used to accept connection requests.<br>If the socket is a datagram socket, the socket address for the communication target to send data to has not been determined. |
| EMSGSIZE | Data exceeds the size of the internal send buffer. (Datagram sockets only.) |
| ENETDOWN | Local network interface is down. |
| ENETUNREACH | Could not reach the connection target. |
| ENOBUFS | Failed to allocate the temporary socket address for the empty socket. Alternatively, failed to allocate the temporary send buffer. |
| ENOTCONN | Specified socket is not connected to the remote host. |
| ETIMEDOUT | The process timed out. |
| EWOULDBLOCK | Same as EAGAIN. |
| EBUSY | The internal process is busy. There may be a shortage of sessions specified by Initialize(). |

## 6.1.9. Setting Options

Call the nn::socket::SetSockOpt or nn::socket::GetSockOpt() functions to set and get socket options, such as internal socket setting values and internal state information.

**Code 6-10. Setting Options**

```
s32 nn::socket::SetSockOpt(s32 s, s32 level, s32 optname, const void* optval,
                           s32 optlen);
s32 nn::socket::GetSockOpt(s32 s, s32 level, int optname, void* optval,
                           int* optlen)
```

For more information about the values that can be specified for these function parameters, see the header file nn/socket/socket_User.autogen.h.

A return value of 0 indicates success. A negative return value indicates an error. The following table lists the errors that may occur.

**Table 6-11. Errors When Setting Options**

| Return Value | Error Description |
|---|---|
| ENETRESET | Library not initialized. |
| | |

| | |
|---|---|
| EBADF | The specified socket descriptor is not valid. |
| EINVAL | Invalid operation. |
| ENETDOWN | Local network interface is down. |
| ENOMEM | Insufficient memory to allocate working memory. |
| ENOPROTOOPT | Unsupported option. |
| EBUSY | The internal process is busy. There may be a shortage of sessions specified by `Initialize()`. |

## 6.1.10. Disconnecting a Socket

Call the `nn::socket::Shutdown()` function to halt socket communications and disconnect a socket.

**Code 6-11. Disconnecting a Socket**

```
s32 nn::socket::Shutdown(s32 s, s32 how);
```

*s* specifies the socket descriptor for the socket to shut down.

*how* specifies how to shut down the socket. Specify `SHUT_RD` to stop receiving data, `SHUT_WR` to stop sending data, and `SHUT_RDWR` to stop both sending and receiving.

A return value of 0 indicates success. A negative return value indicates an error. The following table lists the errors that may occur.

**Table 6-12. Errors When Disconnecting a Socket**

| Return Value | Error Description |
|---|---|
| ENETRESET | Library not initialized. |
| EBADF | The specified socket descriptor is not valid. |
| ENOTCONN | Specified socket is not connected to the remote host. |
| EINVAL | Invalid operation. |
| EBUSY | The internal process is busy. There may be a shortage of sessions specified by `Initialize()`. |

## 6.1.11. Closing a Socket

Only a limited number of sockets may be used at any one time. Close any unneeded sockets by calling the `nn::socket::Close()` function.

**Code 6-12. Closing a Socket**

```
s32 nn::socket::Close(s32 s);
```

The *s* parameter specifies the socket descriptor for the socket to close.

A socket can no longer be used after it has been closed. If a function blocks after being called using a closed socket, the blocks are released and an error is returned. When closing a stream socket in asynchronous mode, the socket is closed according to the `Linger` option setting.

The default behavior for the `Close()` function is to return immediately without blocking. Any remaining data to send is then sent in the background, after which the socket releases the resources it was using.

A return value of 0 indicates success. A negative return value indicates an error. The following table lists the errors that may occur.

**Table 6-13. Errors When Closing a Socket**

| Return Value | Error Description |
|---|---|
| ENETRESET | Library not initialized. |
| EBADF | The specified socket descriptor is not valid. |
| ENOTCONN | Specified socket is not connected to the remote host. |
| EINVAL | Invalid operation. |
| EBUSY | The internal process is busy. There may be a shortage of sessions specified by `Initialize()`. |

## 6.1.12. Finalizing

When no longer using the SOCKET library, call the `nn::socket::Finalize()` function to finalize the library.

**Code 6-13. Finalizing the SOCKET Library**

```
nn::Result nn::socket::Finalize(void);
```

This releases any resources in use by the library and destroys any socket descriptors in use.

## 6.1.13. Utility Functions

The SOCKET library includes various utility functions for purposes such as getting socket

addresses, getting network adapter information, getting remote host information via DNS, converting addresses to and from strings and numeric values, and converting numbers to the host byte order and the network byte order.

**Code 6-14. Getting a Socket Address**

```
s32 nn::socket::GetSockName(s32 s, nn::socket::SockAddrIn* sockAddr);
s32 nn::socket::GetPeerName(s32 s, nn::socket::SockAddrIn* sockAddr);
```

The `nn::socket::GetSockName()` function gets the local socket address for the socket indicated by the socket descriptor passed in *`s`* and puts it in *`sockAddr`*. The local socket address is the communication source socket address. If no local socket address has been configured using the `nn::socket::Bind` or `nn::socket::Connect()` functions, this function returns an address of `0.0.0.0` if called on a UDP-protocol socket (a datagram socket), and returns an error if called on a TCP-protocol socket (a stream socket).

The `nn::socket::GetPeerName()` function gets the remote socket address for the socket indicated by the socket descriptor passed in *`s`* and puts it in *`sockAddr`*. The remote socket address is the communication target socket address.

A return value of 0 indicates success for either function. A negative return value indicates an error. The following table lists the errors that may occur.

**Table 6-14. Errors When Getting a Socket Address**

| Return Value | Error Description |
|---|---|
| ENETRESET | Library not initialized. |
| EBADF | The specified socket descriptor is not valid. |
| ENOTCONN | Not connected to remote host or no communication target. (Only for the `GetPeerName()` function.) |
| EINVAL | Invalid operation. |
| ENETDOWN | Local network interface is down. |
| EBUSY | The internal process is busy. There may be a shortage of sessions specified by `Initialize()`. |

**Code 6-15. Getting Network Adapter Information**

```
u32 nn::socket::GetHostId(void);
```

The `nn::socket::GetHostId()` function returns the IPv4 address of the local host. A return value of 0 indicates that the network cannot be used. The returned IP address is a 32-bit numeric value in network byte order.

**Code 6-16. Getting Remote Host Information via DNS**

```
nn::socket::HostEnt* nn::socket::GetHostByName(const char8* name);
nn::socket::HostEnt* nn::socket::GetHostByAddr(const void* addr, s32 len,
                                               s32 type);
s32 nn::socket::GetAddrInfo(const char8* nodeName, const char8* servName,
                const nn::socket::AddrInfo* hints, nn::socket::AddrInfo** res);
void nn::socket::FreeAddrInfo(nn::socket::AddrInfo* head);
s32 nn::socket::GetNameInfo(const void* sa, char8* node, s32 nodeLen,
                            char8* service, s32 serviceLen, s32 flags);
```

The `nn::socket::GetHostByName()` and `nn::socket::GetHostByAddr()` functions both get information for the specified host. `GetHostByName()` searches for a host based on either the host's name or address string in dot-decimal notation, and `GetHostByAddr()` searches based on the host's address. Processing blocks while searching, and the function may query the DNS server. The actual representation of the structure holding the return value is a buffer internal to the library, so neither of these functions is thread-safe.

`nn::socket::GetAddrInfo` gets information for the host found based on hostname and service name. Configure the search parameters using the `flags` member of the `nn::socket::AddrInfo` structure passed in the *`hints`* parameter. The `flags` member stores a bitwise OR of the flags defined by the `nn::socket::AddrInfoFlag` enumerators.

**Table 6-15. `nn::socket::AddrInfoFlag` Enumerators**

| Flag | Description |
|---|---|
| `AI_PASSIVE` | Uses `0.0.0.0` when *`nodeName`* specifies NULL. |
| `AI_CANONNAME` | Gets the canonical name for *`nodeName`*. |
| `AI_NUMERICHOST` | *`nodeName`* specifies an address in dot-decimal notation. |
| `AI_NUMERICSERV` | *`servName`* specifies a port number as a string. |

Processing blocks while searching, and the function may query the DNS server. The search results structure returned in *`res`* is stored in memory allocated by the library from the allocator specified in the `nn::socket::Initialize()` function. Be sure to release this memory when the search results are no longer needed by calling the `nn::socket::FreeAddrInfo()` function.

A return value of 0 indicates success for the `nn::socket::GetAddrInfo()` function. A negative return value indicates an error. The following table lists the errors that may occur.

**Table 6-16. Errors When Getting Host Information With nn::socket::GetAddrInfo**

| Return Value | Error Description |
|---|---|
| `EAI_BADFLAGS` | The *`nodeName`* is either NULL or an empty string even though `AI_CANONNAME` is specified for *`hints`*. |
| `EAI_FAIL` | An unrecoverable error has occurred. |
| `EAI_FAMILY` | The address family specified in *`hints`* is not supported. |
| `EAI_MEMORY` | Failed to allocate memory. |
| `EAI_NONAME` | Could not find the specified host. |
| | The socket type, socket protocol, and socket combinations specified in *`hints`* are |

| | |
|---|---|
| EAI_SOCKTYPE | not supported. |
| EAI_SYSTEM | A system error occurred. This occurs when this function is called without Initialize is called. |

The nn::socket::GetNameInfo() function searches for the hostname and service name based on an address. Configure the search parameters by specifying a bitwise OR of the flags defined by the nn::socket::NameInfoFlag enumerators in the *flags* parameter.

**Table 6-17. nn::socket::NameInfoFlag Enumerators**

| Flag | Description |
|---|---|
| NI_NOFQDN | Gets only the node name portion of a fully qualified domain name as the hostname. |
| NI_NUMERICHOST | Stores the host address in *node* after converting to a dot-decimal notation string. |
| NI_NAMEREQD | If the specified host cannot be found, the numeric host address is converted to a string, not replaced, and handled as an error (EAI_NONAME). |
| NI_NUMERICSERV | Stores the service (port number) in *service*, converted to a numeric string. |

Processing blocks while searching, and the function may query the DNS server.

A return value of 0 indicates success. A negative return value indicates an error. The following table lists the errors that may occur.

**Table 6-18. Errors When Getting Host Information With nn::socket::GetNameInfo**

| Return Value | Error Description |
|---|---|
| EAI_FAMILY | Unsupported protocol family included in the socket address specified in *sa*. |
| EAI_NONAME | Invalid buffer size specified in *nodeLen* or *serviceLen*.<br>If NI_NAMEREQD was specified in *flags*, the specified host was not found. |
| EAI_SYSTEM | A system error occurred. This occurs when this function is called without Initialize is called. |

**Code 6-17. Converting an Address**

```
s32 nn::socket::InetAtoN(const char* cp, nn::socket::InAddr* inp);
char* nn::socket::InetNtoA(nn::socket::InAddr in);
s32 nn::socket::InetPtoN(int af, const char* src, void* dst);
const char* nn::socket::InetNtoP(int af, const void* src, char* dst,
                                 unsigned len);
```

Call the nn::socket::InetAtoN() and nn::socket::InetNtoA() functions to convert an address between a numeric string and the nn::socket::InAddr structure's numeric format. These functions can convert IPv4 host addresses. The nn::socket::InetPtoN() and nn::socket::InetNtoP() functions previously allowed various values for the af address family argument, but on the CTR-SDK these functions only take the value AF_INET for this argument.

These functions can be used even if `nn::socket::Initialize` has not been called.

**Code 6-18. Converting Byte Order**

```
bit32 nn::socket::HtoNl(bit32 v);
bit32 nn::socket::NtoHl(bit32 v);
bit16 nn::socket::HtoNs(bit16 v);
bit16 nn::socket::NtoHs(bit16 v);
```

Use these functions to convert a numeric value from host byte order to network byte order (`HtoN`) and back again (`NtoH`). The functions ending in "`l`" take 32-bit values, while those ending in "`s`" take 16-bit values.

# 6.2. SSL Communication

The SSL library is a wrapper for SSL communication to help keep communication secure.

## 6.2.1. Initializing

Call the `nn::ssl::Initialize()` function to initialize the SSL library.

**Code 6-19. Initializing the SSL Library**

```
nn::Result nn::ssl::Initialize(void);
```

## 6.2.2. Generating an Instance of the Connection Class

SSL communication is handled by the `nn::ssl::Connection` class, which itself is a wrapper for sockets.

**Code 6-20. Connection Class Constructor and Assigning Sockets**

```
class nn::ssl::Connection
{
    explicit Connection(s32 socket);
    explicit Connection();
    bool AssignSocket(s32 socket);
}
```

Two constructors are available: one that takes a socket descriptor as an argument, and one that does not. For `Connection` class instances generated without an argument, you must call the `AssignSocket` member function to bind a socket.

## 6.2.1. Setting the Communication Target

Use the `Initialize` member function to set the communication target server to the connection class.

**Code 6-21. Setting the Communication Target**

```
class nn::ssl::Connection
{
    nn::Result Initialize(const char* pServerName,
                          u32 verifyOpt = VERIFY_NONE);
}
```

*pServerName* specifies the hostname of the communication target server. When using SSL communication, this hostname is compared against the `dnsName/ipAddress` values in the `subjectAltName` extended region or the `CommonName` of the server certificate, and a communication error occurs if these values do not match. The maximum length of the specifiable hostname string, including the string terminator, is `nn::socket::MAXDNAME` bytes.

*verifyOpt* specifies any options for server verification during SSL communication. Omit this parameter to use the default server verification. When implementing a server verification option that is not implemented by default, specify in *verifyOpt* an enumerator value from the `nn::ssl:VerifyOption` enumerated type that specifies this option. To use multiple verification options simultaneously, specify the bitwise OR of these values. This parameter can take the following values.

**Table 6-19. Specifiable Values for Server Verification Options**

| Value | Description |
|---|---|
| `VERIFY_DATE` | Verify the certificate's expiration date. |
| `USE_SESSION_CACHE` | Use SSL resumption and attempt to reuse the session when connecting to the same host multiple times in a row. |
| `VERIFY_EV` | Verifies the EV certificate. Verification succeeds unless the server certificate is linked to an EV certificate.<br>Trusted EV certificates can be set using `AddEVPolicyID`. |
| `VERIFY_IGNORE` | Verifies the server certificate, but ignores results and proceeds. Call `GetCertVerifyErrors` to get verification results. |
| `GET_ALL_SERVER_CERT_CHAIN` | Saves all certificate data in the certificate chain when saving the server certificate in the buffer specified by `SetServerCertBuffer`. Omit this option to save only the server certificate. |

Other options not in the table are implemented by default, as defined in the `nn::ssl::VerifyOption` enumerated type.

> **Note:** When saving all certificates in the certificate chain, each certificate is saved with the first four bytes indicating the length of the certificate data, followed by the certificate data itself. The data layout starts with the first certificate in the chain (the server certificate) and continues through to the last certificate in the chain (the CA certificate). However, if CA verification fails, the CA certificate cannot be designated and is excluded.

## 6.2.4. Setting the Certificate and CRL

Use the following function to configure the connection class with the certificate and CRL stores and the client certificate to use in the handshake with the server.

**Code 6-22. Setting the Certificate and CRL**

```
class nn::ssl::Connection
{
    nn::Result SetServerCertStore(nn::ssl::CertStore& certStore);
    nn::Result SetClientCert(nn::ssl::ClientCert& clientCert);
    nn::Result SetCRLStore(nn::ssl::CrlStore& crlStore);
}
```

Use the `SetServerCertStore()` function to set a server certificate store, use `SetCRLStore` to set a CRL store, and use `SetClientCert` to set a client certificate.

### 6.2.4.1. Certificate Store Class

The certificate store class is defined by `nn::ssl::CertStore`.

**Code 6-23. Certificate Store Class**

```
class nn::ssl::CertStore
{
    explicit CertStore();
    virtual ~CertStore(void);

    nn::Result Initialize(void);
    nn::Result Finalize(void);

    nn::Result RegisterCert(const u8* pCertData, size_t certDataSize,
                            nn::ssl::CertId* pCertIdCourier=NULL);
    nn::Result RegisterCert(nn::ssl::InternalCaCert inCaCertName,
                            nn::ssl::CertId* pCertIdCourier=NULL);
    nn::Result UnRegisterCert(nn::ssl::CertId certId);
}
```

The constructor has no arguments. Generate an instance and call the `Initialize()` function.

Call the `RegisterCert` member function to register a certificate to the certificate store. *`pCertData`* and *`certDataSize`* specify the certificate data and its size in bytes. Specify `nn::ssl::InternalCaCert` as the argument instead to set a certificate stored internally to the certificate store. *`pCertIdCourier`* may be omitted unless you need to unregister an individual certificate. Call the `Finalize()` function to unregister all certificates that have not been individually unregistered. Call the `RegisterCert` member function multiple times to register multiple certificates to the certificate store.

Use the `UnRegisterCert` member function to unregister an individual certificate from the certificate store. Specify the certificate ID obtained in *`pCertIdCourier`* when registering the certificate via `RegisterCert` as the value for the *`certId`* argument.

Be sure to call the `Finalize()` function when the certificate store is no longer needed.

## 6.2.4.2. CRL Store Class

The CRL store class is defined by `nn::ssl::CrlStore`.

**Code 6-24. CRL Store Class**

```
class nn::ssl::CrlStore
{
    explicit CrlStore();
    virtual ~CrlStore(void);

    nn::Result Initialize(void);
    nn::Result Finalize(void);

    nn::Result RegisterCrl(const u8* pCrlData, size_t crlDatasize,
                           nn::ssl::CrlId* pCrlIdCourier=NULL);
    nn::Result RegisterCrl(nn::ssl::InternalCrl inCrlName,
                           nn::ssl::CrlId* pCrlIdCourier=NULL);
    nn::Result UnRegisterCrl(nn::ssl::CrlId crlId);
}
```

The constructor has no arguments. Generate an instance and call the `Initialize()` function.

Call the `RegisterCrl` member function to register a CRL to the CRL store. *`pCrlData`* and *`crlDataSize`* specify the CRL data and its size in bytes. Specify `nn::ssl::InternalCrl` as the argument instead to set a CRL stored internally to the CRL store. *`pCrlIdCourier`* may be omitted unless you need to unregister an individual CRL. Call the `Finalize()` function to unregister all CRLs that have not been individually unregistered. Call the `RegisterCrl` member function multiple times to register multiple CRLs to the CRL store.

Use the `UnRegisterCrl` member function to unregister an individual CRL from the CRL store. Specify the CRL ID obtained in *`pCrlIdCourier`* when registering the certificate via `RegisterCrl` as the value for the *`crlId`* argument.

Be sure to call the `Finalize()` function when the CRL store is no longer needed.

### 6.2.4.3. Client Certificate Class

The client certificate class is defined by `nn::ssl::ClientCert`.

**Code 6-25. Client Certificate Class**

```
class nn::ssl::ClientCert
{
    explicit ClientCert();
    virtual ~ClientCert(void);

    nn::Result Initialize(const u8* pCertData, size_t certDataSize,
                          const u8* pPrivateKeyData, size_t
privateKeyDataSize);
    nn::Result Initialize(nn::ssl::InternalClientCert inClientCertName);
    nn::Result Finalize(void);
}
```

The constructor has no arguments. Generate an instance and call the `Initialize()` function.

*`pCertData`* and *`certDataSize`* specify the client certificate data and its size in bytes. *`pPrivateKeyData`* and *`privateKeyDataSize`* specify the private key data and its size in bytes. Specify `nn::ssl::InternalClientCert` instead to use an internal client certificate.

Be sure to call the `Finalize()` function when the client certificate is no longer needed.

## 6.2.5. Handshaking

Call the `DoHandshake()` function of the `Connection` class to use the configured certificate and CRL stores and client certificate to handshake with the connection target server.

**Code 6-26. Handshaking**

```
class nn::ssl::Connection
{
    nn::Result SetServerCertBuffer(uptr bufferAddress, size_t bufferSize);
    nn::Result DoHandshake(void);
    nn::Result DoHandshake(size_t* pServerCertSize,
                           u32*    pServerCertNum = NULL);
}
```

Call the `DoHandshake()` function version with arguments when you need information about a server certificate sent from the server. The size of the server certificate in bytes is stored in the variable pointer specified in *`pServerCertSize`*.

Specify the buffer to store the server certificate by calling the `SetServerCertBuffer()` function. The buffer specified must be allocated from outside of device memory. The starting address of the buffer specified must be aligned to `nn::ssl::Connection::BUFFER_ALIGNMENT` (4,096 bytes), and the size of the buffer must be a multiple of `nn::ssl::Connection::BUFFER_UNITSIZE` (4,096 bytes). Calling the `DoHandshake()` function version with arguments without specifying a

buffer via `SetServerCertBuffer` is effectively identical to calling the `DoHandshake()` function version that has no arguments.

## 6.2.6. Transferring Data

After confirming a successful handshake, you can start sending and receiving data via SSL communication.

**Code 6-27. Sending and Receiving Data via an SSL Connection**

```
class nn::ssl::Connection
{
    nn::Result Read(u8* pDataBuf, size_t dataBufSize,
                    size_t* pReadSizeCourier = NULL);
    nn::Result Peek(u8* pDataBuf, size_t dataBufSize,
                    size_t* pReadSizeCourier = NULL);
    nn::Result Write(const u8* pDataBuf, size_t dataBufSize,
                     size_t* pWrittenDataSizeCourier = NULL);
}
```

The `Read()` and `Peek()` member functions both receive data. `Peek` does not change the state of data received, and data received this way can be received again. *`pDataBuf`* and *`dataBufSize`* specify a buffer to receive the data and its size in bytes. Specify a pointer to a variable in *`pReadSizeCourier`* for receiving the data size if you need the size in bytes of the received data.

The `Write()` member function sends data. *`pDataBuf`* and *`dataBufSize`* specify a buffer to store the data to send and its size in bytes. Specify a pointer to a variable in *`pWrittenDataSizeCourier`* for receiving the data size if you need the size in bytes of the sent data.

## 6.2.7. Disconnecting

Only a limited number of SSL connections may be used at any one time. Be sure to call the `Shutdown` member function on class instances that are no longer needed to disconnect SSL connections and finalize the instances.

**Code 6-28. Disconnecting an SSL Connection**

```
class nn::ssl::Connection
{
    nn::Result Shutdown(void);
}
```

### 6.2.8. Finalizing

When you are no longer using the SSL library, call the `nn::ssl::Finalize()` function to finalize the library.

**Code 6-29. Finalizing the SSL Library**

```
nn::Result nn::ssl::Finalize(void);
```

## 6.3. HTTP Communication

The HTTP library handles connecting to the specified URL and communicating over the network via the HTTP protocol. The library also supports the HTTPS protocol.

### 6.3.1. Initializing

Call the `nn::http::Initialize()` function to initialize the HTTP library.

**Code 6-30. Initializing the HTTP Library**

```
nn::Result nn::http::Initialize(uptr bufferAddress = 0, size_t bufferSize = 0);
```

You must call this function before using the HTTP library. The function returns `nn::http::ResultAlreadyInitialized` if called after the library is already initialized.

Specify *`bufferAddress`* and *`bufferSize`*. The buffer you pass for communications **must be allocated from outside of device memory**. The starting address of the buffer must be aligned to `nn::http::BUFFER_ALIGNMENT` (4,096 bytes), and the size of the buffer must be a multiple of `nn::http::BUFFER_UNITSIZE` (4,096 bytes).

### 6.3.2. Generating an Instance of the Connection Class

Use the `nn::http::Connection` class to work with HTTP connections. You must generate one instance of this class for each URL to connect to. The number of connections that can be used at one time by the system as a whole and by the application is limited, so take care when managing connections.

There are two constructor versions: one with arguments and one without. After generating an instance using the constructor with no arguments, you must call the `Initialize` member function to configure the URL to connect to, the connection method, and whether to use a default proxy.

**Code 6-31. HTTP Connection Class Constructors and Initialization**

```
class nn::http::Connection
{
    explicit Connection(void);
    explicit Connection(const char* pUrl,
                        RequestMethod method = REQUEST_METHOD_GET,
                        bool isUseDefaultProxy = true);
    nn::Result Initialize(const char* pUrl,
                          RequestMethod method = REQUEST_METHOD_GET,
                          bool isUseDefaultProxy = true);
}
```

Specify the URL to connect to in the *`pUrl`* parameter.

Specify the HTTP request method in the *`method`* parameter. Select from one of the following three types.

**Table 6-20. List of Method Types**

| Definition | Description |
|---|---|
| `REQUEST_METHOD_GET` | GET method. The parameters indicating which data to get are embedded in the URL. |
| `REQUEST_METHOD_POST` | POST method. The data to send to the server is added as POST data in the body of the request. |
| `REQUEST_METHOD_HEAD` | HEAD method. Similar to the GET method, but only retrieves the message header for the specified URL. |

Specify `true` for the *`isUseDefaultProxy`* parameter to use the default proxy when connecting. Specify `false` to use a different proxy, and configure that separately.

In case of any error occurring during initialization, check the `Description` property of the `nn::Result` class instance returned by the `Initialize()` function.

**Table 6-21. Errors Occurring During Initialization**

| Description Value | Cause of Error |
|---|---|
| `ER_CONN_ADD` | The system as a whole has no free communications resources. |
| `ER_REQ_URL` | The specified URL is invalid. |
| `ER_ALREADY_ASSIGN_HOST` | A destination URL has already been specified |
| `ER_CONN_PROCESS_MAX` | No free application-allocated communications resources. |

## 6.3.3. Configuring Communication

Use these functions to configure various aspects of communication, such as proxy settings, basic verification, or the receive buffer size of the socket used in communication. These settings must be configured before initiating a connection.

**Code 6-32. Configuring Communication**

```
class nn::http::Connection
{
    nn::Result SetProxy(const char* pProxyName, u16 port,
                        const char* pUserName, const char* pPassword);
    nn::Result SetBasicAuthorization(const char* pUserName,
                                     const char* pPassword);
}
```

Call the `SetProxy()` function to use a proxy other than the default. Specify the proxy URL and port in the *`pProxyName`* and *`port`* parameters, and specify the verification user name and password in the *`pUserName`* and *`pPassword`* parameters.

Call the `SetBasicAuthorization()` function if you need to configure basic user verification on connection. Specify the user name and password to use in the *`pUserName`* and *`pPassword`* parameters.

## 6.3.3.1. Configuring for HTTPS Communication

To use the HTTPS protocol, you must first configure SSL communication settings, such as the CA, CRL, and client certificates.

**Code 6-33. Configuring HTTPS Communication**

```
class nn::http::Connection
{
    nn::Result SetRootCa(const u8* pCertData, size_t certDataSize);
    nn::Result SetRootCa(nn::http::InternalCaCertId inCaCertName);
    nn::Result SetRootCaStore(nn::http::CertStore& certStore);
    nn::Result SetCrl(const u8* pCrlData, size_t crlDataSize);
    nn::Result SetCrl(nn::http::InternalCrlId inCrlName);
    nn::Result SetCrlStore(CrlStore& crlStore);
    nn::Result SetClientCert(const u8* pCertData, size_t certDataSize,
                             const u8* pPrivateKeyData, size_t
privateKeyDataSize);
    nn::Result SetClientCert(nn::http::InternalClientCertId inClientCertName);
    nn::Result SetClientCert(nn::http::ClientCert& clientCert);
    nn::Result GetSslError(s32* pResultCodeBuf) const;
    nn::Result SetVerifyOption(u32 verifyOption);
    nn::Result DisableVerifyOptionForDebug(u32 excludeVerifyOptions);
}
```

Call the `SetRootCa()` function to register a CA certificate. You can register the certificate data directly, or specify `nn::http::InternalCaCertId` to use the internal CA certificate included in the system.

If you need to register multiple CA certificates, either call `SetRootCa()` multiple times, or call

`SetRootCaStore()` and specify an instance of the `nn::http::CerStore` class as an argument. This allows you to use the same set of CA certificates for multiple connections.

Do not destroy the instance of the `nn::http::CertStore` class until you have finished using the set of CA certificates.

**Code 6-34. nn::http::CertStore Class Definition**

```
class nn::http::CertStore
{
    explicit CertStore();
    virtual ~CertStore (void);

    nn::Result Initialize(void);
    nn::Result Finalize(void);
    nn::Result RegisterCert(const u8* pCertData, size_t certDataSize,
                            nn::http::CertId* pCertIdCourier=NULL);
    nn::Result RegisterCert(nn::http::InternalCaCertId inCaCertName,
                            nn::http::CertId* pCertIdCourier=NULL);
    nn::Result UnRegisterCert(nn::http::CertId certId);
}
```

Be sure to call the `Initialize` member function when using an `nn::http::CertStore` class instance. Call the `RegisterCert()` function to register a CA certificate. Call the function multiple times to register multiple CA certificates to use them together as a set.

To register an individual CA certificate, call the `UnRegisterCert()` function, passing as an argument the value returned in the *`pCertIdCourier`* parameter when you registered the certificate. After you finish using a class instance, you must call the `Finalize()` function to destroy the instance.

Call the `SetCrl()` function to set a CRL. You can set the CRL data directly, or specify `nn::http::InternalCrlId` to use the internal CA certificates included in the system. Note that there are no internal CRLs at present.

Much as for CA certificates, if you need to register multiple CRLs, either call `SetCrl()` multiple times, or call `SetCrlStore()` and specify an instance of the `nn::http::CrlStore` class as an argument. Use the `nn::http::CrlStore` class the same way you would use the `nn::http::CertStore` class.

**Code 6-35. nn::http::CrlStore Class Definition**

```
class nn::http::CrlStore
{
    explicit CrlStore();
    virtual ~CrlStore (void);

    nn::Result Initialize(void);
    nn::Result Finalize(void);
    nn::Result RegisterCrl(const u8* pCrlData, size_t crlDataSize,
                            nn::http::CrlId* pCrlIdCourier=NULL);
    nn::Result RegisterCrl(nn::http::InternalCrlId inCrlName,
                            nn::http::CrlId* pCrlIdCourier=NULL);
    nn::Result UnRegisterCrl(nn::http::CrlId crlId);
}
```

Call the `SetClientCert()` function to register a client certificate. You can configure the

certificate by specifying the certificate data and secret key, by specifying the system's internal client certificate, or by using an `nn::http::ClientCert` class instance to use the same set of client certificates for multiple connections.

When using an `nn::http::ClientCert` class instance with multiple connections, do not destroy the instance until you finish using it.

**Code 6-36. nn::http::ClientCert Class Definition**

```
class nn::http::ClientCert
{
    explicit ClientCert();
    virtual ~ClientCert (void);

    nn::Result Initialize(const u8* pCertData, size_t certDataSize,
                          const u8* pPrivateKeyData, size_t
privateKeyDataSize);
    nn::Result Initialize(nn::http::InternalClientCertId inClientCertName);
    nn::Result Finalize(void);
}
```

The `nn::http::ClientCert` class includes two versions of its `Initialize()` function, one that allows you to specify the certificate data and secret key, and one that uses the system's internal client certificate. After you finish using a class instance, you must call the `Finalize()` function to destroy the instance.

Call the `SetVerifyOption()` function to enable any server verification option that is disabled by default. The options enabled by default are `VERIFY_COMMON_NAME` (`CommonName` verification), `VERIFY_ROOT_CA` (RootCA verification), and `VERIFY_SUBJECT_ALT_NAME` (`SubjectAlternativeName` verification).

Other options that you can enable in the current SDK version are `VERIFY_DATE` (checks for certificate expiration), `USE_SESSION_CACHE` (allows for sessions to be resumed), and `VERIFY_EV` (EV certificate verification).

You can call the `DisableVerifyOptionForDebug()` function to disable all verification options, but note that this is intended only for debugging purposes.

Check for any errors occurring during HTTPS communication by calling the `GetSslError()` function and examining the value stored in the *`pResultCodeBuf`* parameter. These error code values are defined by the `nn::ssl::ResultCode` class.

## 6.3.4. Configuring Data to Send

Before initiating a connection and sending data, you must first add a message field header or POST data. Use the lazy POST data setting mode to add POST data after initiating a connection.

**Code 6-37. Adding a Header or Data Before Sending**

```
class nn::http::Connection
{
```

```
    nn::Result AddHeaderField(const char* pLabel, const char* pValue);
    nn::Result SetPostDataEncoding(nn::http::EncodingType type);
    nn::Result AddPostDataAscii(const char* pLabel, const char* pValue);
    nn::Result AddPostDataBinary(const char* pLabel, const void* pValue,
                                 size_t valueSize);
    nn::Result AddPostDataRaw(const void* pValue, size_t valueSize);
}
```

Call the AddHeaderField() function to add a header to the message field. For example, specify the strings Connection for the *pLabel* parameter and keep-alive for the *pValue* parameter to add a header with the label Connection and the content keep-alive.

Call the SetPostDataEncoding() function to specify the POST data encoding method, and then call AddPostDataAscii or another appropriate function to add the data.

Specify a value defined by the nn::http::EncodingType enumerated type for the *type* argument.

**Table 6-22. POST Data Encoding Methods**

| Definition | Description |
|---|---|
| ENCODING_TYPE_AUTO | Sends data in URL encoding if all data was added using the AddPostDataAscii() function. Sends as multipart if any data was added using the AddPostDataBinary() function. (Default.) |
| ENCODING_TYPE_URL | Sends data in URL encoding. Call the AddPostDataAscii() function to add POST data. |
| ENCODING_TYPE_MULTIPART | Sends data without encoding it. POST data added using the AddPostDataAscii() function is not encoded. |

POST data added using the AddPostDataAscii() function (ASCII string data) and sent in URL encoding has both labels and data encoded. POST data that includes binary data or for which the encoding method is specified as multi-part is sent without encoding either the labels or data.

POST data added using the AddPostDataBinary() function (binary data) is sent as is, without encoding.

Call the AddPostDataRaw() function to add multiple POST data items at the same time, unlike the two functions above. This function only allows you to specify binary POST data. Calling either the AddPostDataAscii or AddPostDataBinary() functions after calling AddPostDataRaw returns an error with ER_POST_ADDED_ANOTHER set for the Description. Calling AddPostDataRaw a second time discards the data added the first time, adding instead the POST data specified in this second call.

## 6.3.4.1. Lazy POST Data Setting Mode

Use the lazy POST data setting mode to send POST data after initiating a connection. Note that the SendPostData*() functions execute synchronously when sending POST data in this mode, so processing blocks until sending is finished. However, you can impose a timeout by specifying a

value for the *timeout* parameter of the overloaded version. This causes the function to call Cancel internally after the timeout period has passed, canceling the connection.

**Code 6-38. Sending in the Lazy POST Data Setting Mode**

```
class nn::http::Connection
{
    nn::Result SetLazyPostDataSetting(nn::http::PostDataType dataType);
    nn::Result NotifyFinishSendPostData(void);
    nn::Result SendPostDataAscii(const char* pLabel, const char* pValue);
    nn::Result SendPostDataAscii(const char* pLabel, const char* pValue,
                                 const nn::fnd::TimeSpan& timeout);
    nn::Result SendPostDataBinary(
                 const char* pLabel, const void* pValue, size_t valueSize);
    nn::Result SendPostDataBinary(
                 const char* pLabel, const void* pValue, size_t valueSize,
                 const nn::fnd::TimeSpan& timeout);
    nn::Result SendPostDataRaw(const void* pValue, size_t valueSize);
    nn::Result SendPostDataRaw(const void* pValue, size_t valueSize,
                               const nn::fnd::TimeSpan& timeout);
}
```

Call the SetLazyPostDataSetting() function to use this mode. The function to call and the sending method used for the POST data differ depending on the value specified for the sending data type specified in *dataType*. Specify a value defined by the nn::http::PostDataType enumerated type for the *dataType* parameter.

**Table 6-23. Sending Data Types in the Lazy POST Data Setting Mode**

| Definition | Description |
|---|---|
| POST_DATA_TYPE_URLENCODE | Sends data in URL encoding. |
| POST_DATA_TYPE_MULTIPART | Sends data in multi-part format. |
| POST_DATA_TYPE_RAW | Sends data in raw format. |

When sending in URL encoding, you can call either the SendPostDataAscii or SendPostDataBinary() functions. Call SendPostDataAscii to send the data in URL encoding with a format where (*label name*)=(*data content*), and call SendPostDataBinary to send identical but unencoded data.

When sending in multi-part format, you can call either the SendPostDataAscii or SendPostDataBinary() functions. Data is sent in chunks, with each chunk starting with boundary data and header fields. Both functions add a header field after the boundary data with the label Content-Disposition: form-data; name=[label], but the SendPostDataBinary() function precedes this header field with another one containing the text Content-Type: application/octet-stream\r\nContent-Transfer-Encoding: binary\r\n.

When sending data in RAW format. You can only call the SendPostDataRaw() function. Data is sent in chunks as chunked data.

You must call the NotifyFinishSendPostData() function after you finish sending all POST data. After calling this function, the system waits to receive an HTTP response.

## 6.3.5. Opening a Connection

Initiate an HTTP connection to the URL specified in initialization. There are two ways to start a connection. Call either the `Connect()` function, which blocks until a connection is established, or the `ConnectAsync()` function, which returns immediately. Call the `Cancel()` function to cancel a connection.

**Code 6-39. Initiating and Canceling a Connection**

```
class nn::http::Connection
{
    nn::Result Connect(void);
    nn::Result ConnectAsync(void);
    nn::Result Cancel(void);
}
```

`Connect` still blocks even if all HTTP connections available to the system are already in use, whereas `ConnectAsync` returns an error immediately.

Call the `Cancel()` function to cancel an HTTP connection, with that instance of the `Connection` class halting communication.

**Warning:** When a connection is initiated using the `nn::http::Connection::Connect` or `nn::http::Connection::ConnectAsync()` function, if the WAN cable to the access point is removed while the function is receiving the header information, the function continues to wait forever.

To prevent this situation from occurring, you must implement your program in a way that terminates communication (by calling `nn::http::Connection::Cancel`) after a timeout period.

## 6.3.6. Receiving a Response

After calling `Connect` or a similar function to start the connection, call the `Read()` function to get the HTTP response message body, `GetStatusCode` to get the response status, `GetHeaderField` to get a specific message header, and `GetHeaderAll` to get all headers. You can impose a timeout by specifying a value for the *`timeout`* parameter of the overloaded version. This causes the function to call `Cancel` internally after the timeout period has passed, canceling the connection.

**Code 6-40. Receiving a Response**

```
class nn::http::Connection
{
    nn::Result Read(u8* pBodyBuf, size_t bufLen);
    nn::Result Read(u8* pBodyBuf, size_t bufLen,
                    const nn::fnd::TimeSpan& timeout);
```

```
    nn::Result GetStatusCode(s32* pStatusCodeCourier) const;
    nn::Result GetStatusCode(s32* pStatusCodeCourier,
                             const nn::fnd::TimeSpan& timeout) const;
    nn::Result GetHeaderField(
                 const char* pLabel, char* pFieldBuf, size_t bufSize,
                 size_t* pFieldLengthCourier = NULL) const;
    nn::Result GetHeaderField(
                 const char* pLabel, char* pFieldBuf, size_t bufSize,
                 const nn::fnd::TimeSpan& timeout,
                 size_t* pFieldLengthCourier = NULL) const;
    nn::Result GetHeaderAll(
                 char* pHeaderBuf, size_t bufSize,
                 size_t* pLengthCourier = NULL) const;
    nn::Result GetHeaderAll(
                 char* pHeaderBuf, size_t bufSize,
                 const nn::fnd::TimeSpan& timeout,
                 size_t* pLengthCourier = NULL) const;
}
```

When calling the `Read()` function, specify the buffer to store the message body in the *`pBodyBuf`* parameter and its size in the *`bufLen`* parameter. If the message body is too big to fit in the buffer, the function stores as much of the body as fits and then returns `nn::http::ResultBodyBufShortage`, in which case you can call `Read` again to read the remainder of the message body. Any other error returned indicates a failure to receive the HTTP response. However, HTTP status errors for events such as authentication failures are sent with the error message as the message body, so that `Read` operates as normal, as a successful connection, returning `nn::http::ResultBodyBufShortage` if the message body is too big to fit in the buffer, and returning an instance whose `IsSuccess` member returns `true` upon reading through to the end of the message body.

When calling the `GetStatusCode()` function, specify an `s32` variable in the *`pStatusCodeCourier`* parameter to store the response status. Check this response status to find out if any communication errors have occurred. If called when receiving the message header, this function blocks until the header has been fully received. If called after calling `Read`, the header has already been fully received, and this function returns immediately.

When calling the `GetHeaderField()` function, specify the label of the header field to get in the *`pLabel`* parameter, the buffer to store this data in the *`pFieldBuf`* parameter, and the buffer size in the *`bufSize`* parameter. As long as *`pFieldBuf`* is not NULL, *`bufSize`* is not `0`, and *`pFieldLengthCourier`* is not NULL, the size in bytes of the specified label's header field is stored in the variable specified in *`pFieldLengthCourier`*.

When calling the `GetHeaderAll()` function, specify the buffer to store all header fields in the *`pHeaderBuf`* parameter and the buffer size in the *`bufSize`* parameter. As long as *`pHeaderBuf`* is not NULL, *`bufSize`* is not `0`, and *`pLengthCourier`* is not NULL, the size in bytes of the buffer required to store all message headers is stored in the variable specified in *`pLengthCourier`*. If called when receiving the message header, these functions block until the header has been fully received. If called after calling `Read`, the header has already been fully received, and these functions return immediately.

## 6.3.7. Getting Connection State and Other Information

The `Connection` class includes functions to get the state of the class, the codes of any errors that have occurred, and response reception progress.

**Code 6-41. Getting Connection State and Other Information**

```
class nn::http::Connection
{
    nn::Result GetStatus(nn::http::Status* pStatusBuf) const;
    nn::Result GetError(nn::http::ResultCode* pResultCodeBuf) const;
    nn::Result GetProgress(size_t* pReceivedLen, size_t* pContentLen) const;
}
```

When calling the `GetStatus()` function, specify a buffer in the *`pStatusBuf`* parameter to store the connection state. The stored value is an `nn::http::Status` enumerated type.

When calling the `GetError()` function, specify a buffer in *`pResultCodeBuf`* parameter to store the error code. The stored value is an `nn::http::ResultCode` enumerated type.

When calling the `GetProgress()` function, specify a variable in the *`pReceivedLen`* parameter to store the size in bytes of the message body received so far, and specify a variable in the *`pContentLen`* parameter to store the total length of the message body. This total length is the value of the `Content-Length` HTTP response header, so *`pContentLen`* stores `0` if the `Content-Length` header is missing.

## 6.3.8. Closing a Connection

After fully receiving the HTTP response and after the class instance is no longer needed, make sure that you call the `Finalize()` function to destroy it. You do not need to call the function for instances that have not been initialized and do not specify a target URL.

**Code 6-42. Closing a Connection**

```
class nn::http::Connection
{
    nn::Result Finalize(void);
}
```

## 6.3.9. Finalizing

Call `nn::http::Finalize` after you finish using the HTTP library. The function returns `nn::http::ResultNotInitializedErr` if the library has not been initialized.

**Code 6-43. Finalizing the HTTP Library**

```
nn::Result nn::http::Finalize(void);
```

# 7. Appendix: Performance of Local Communications

This chapter presents the results of measuring the throughput for local communications. Use this information as a guide when evaluating the throughput for the titles you are developing. Throughput declines when the processing load is high and the signal environment is poor.

## 7.1. Measurement Environment and Measurement Results

Three test units were placed in a shielded box and connected together, with one configured as the master and the other two acting as clients. One client was configured to send but not receive data, and the other was configured only to receive data

To measure throughput, the sample program `SampleDemos/uds/simple` was modified for measurement purposes and compiled using CTR-SDK 4.2.1.

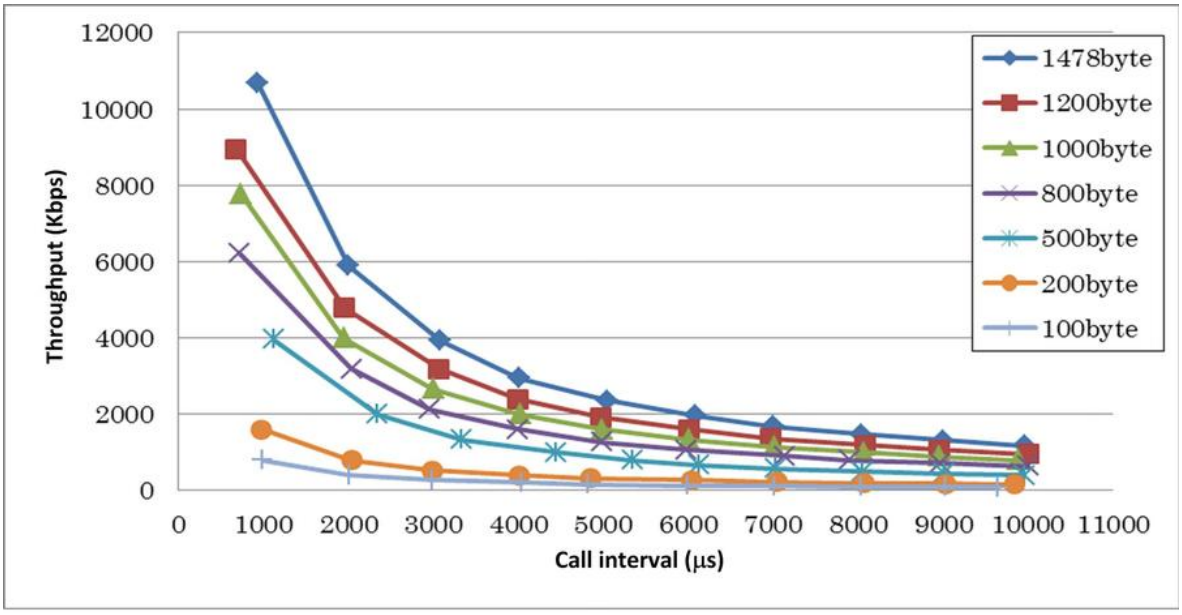
The client configured to only send data called the `SendTo()` function using the combinations of elements shown in the table below.

**Table 7-1. Elements Used for Measuring Performance of Local Communications**

| Element | Conditions |
|---|---|
| Data sizes | 100 / 200 / 500 / 800 / 1000 / 1200 / 1478 bytes |
| Call intervals | 1000 / 2000 / 3000 / 4000 / 5000 / 6000 / 7000 / 8000 / 9000 / 10000 μs |
| Send options | 0x00 / NO_WAIT / FORCE_DIRECT_BC / NO_WAIT \| FORCE_DIRECT_BC |

The measurement results are presented in the graphs below, which give the throughput and the error rate for the different send options, as measured on the client that was configured only to receive data.

**Figure 7-1. Send Option = 0x00**



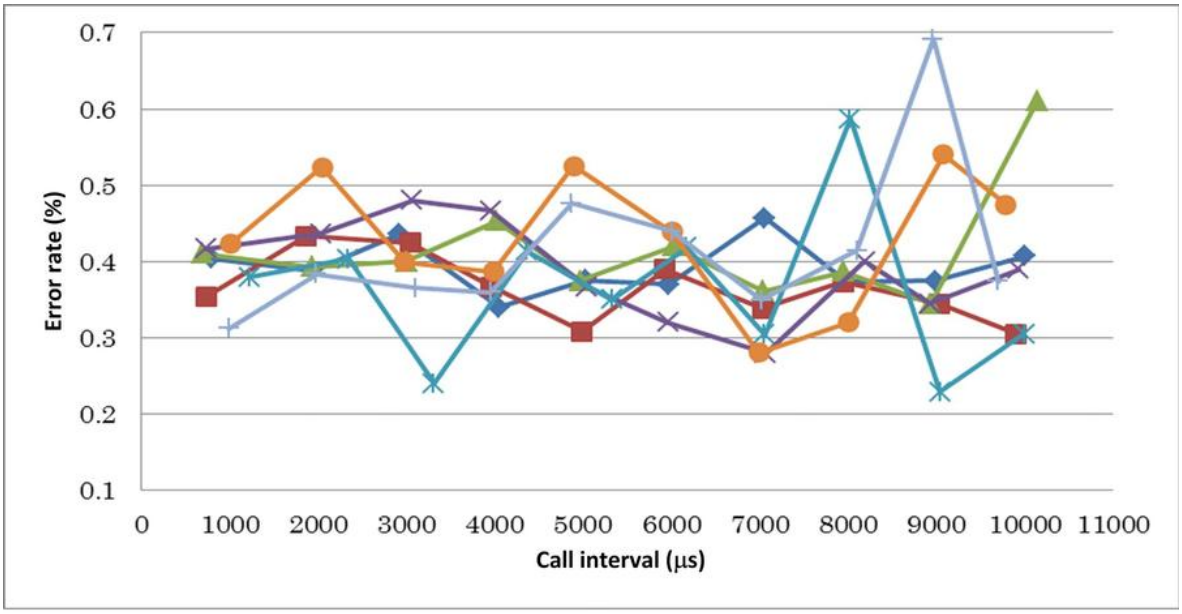
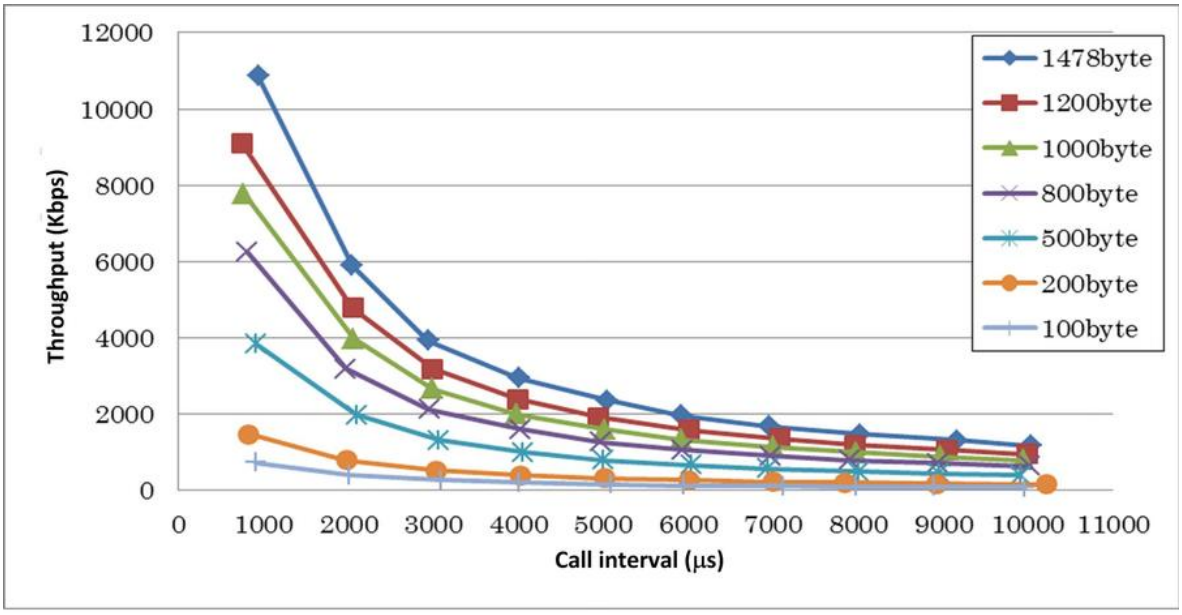




**Figure 7-2. Send Option = NO_WAIT**



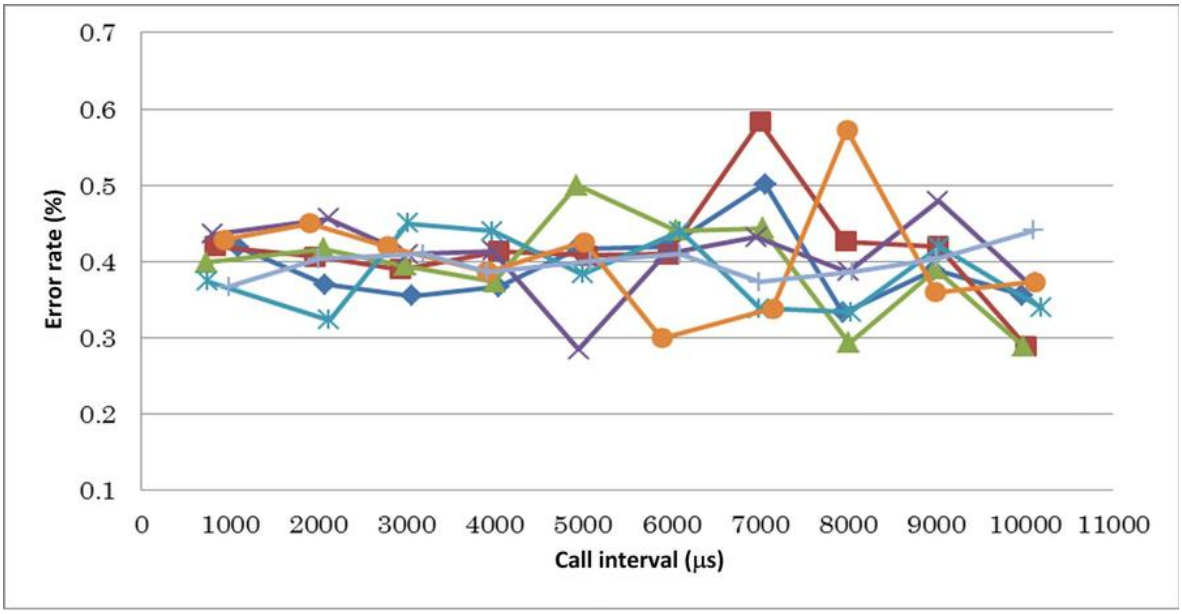
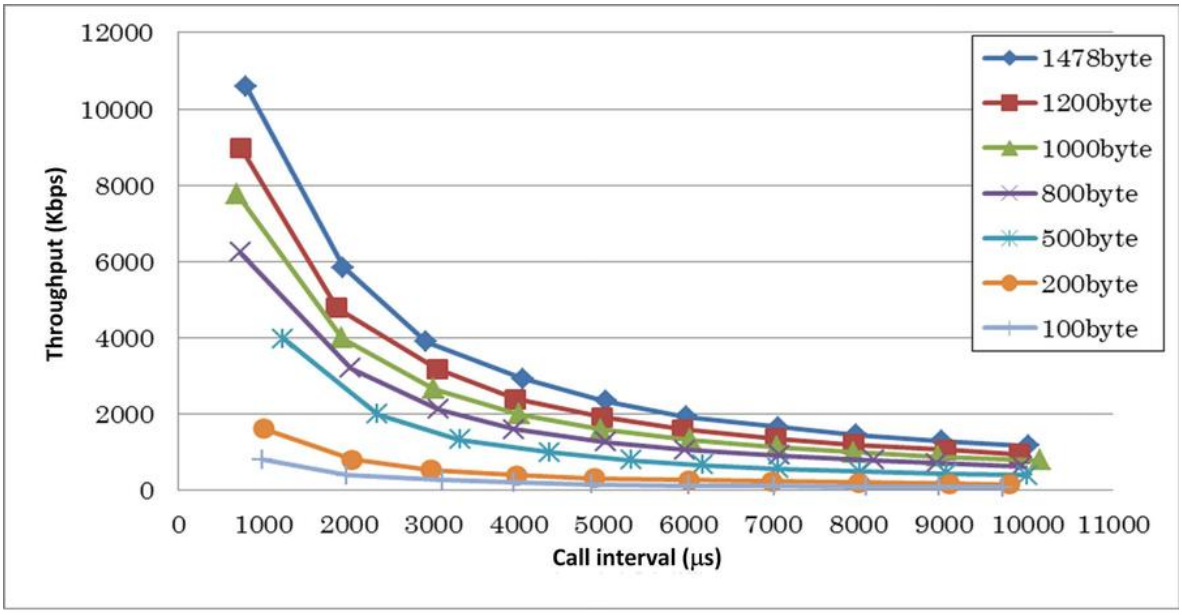




**Figure 7-3. Send Option = FORCE_DIRECT_BC**



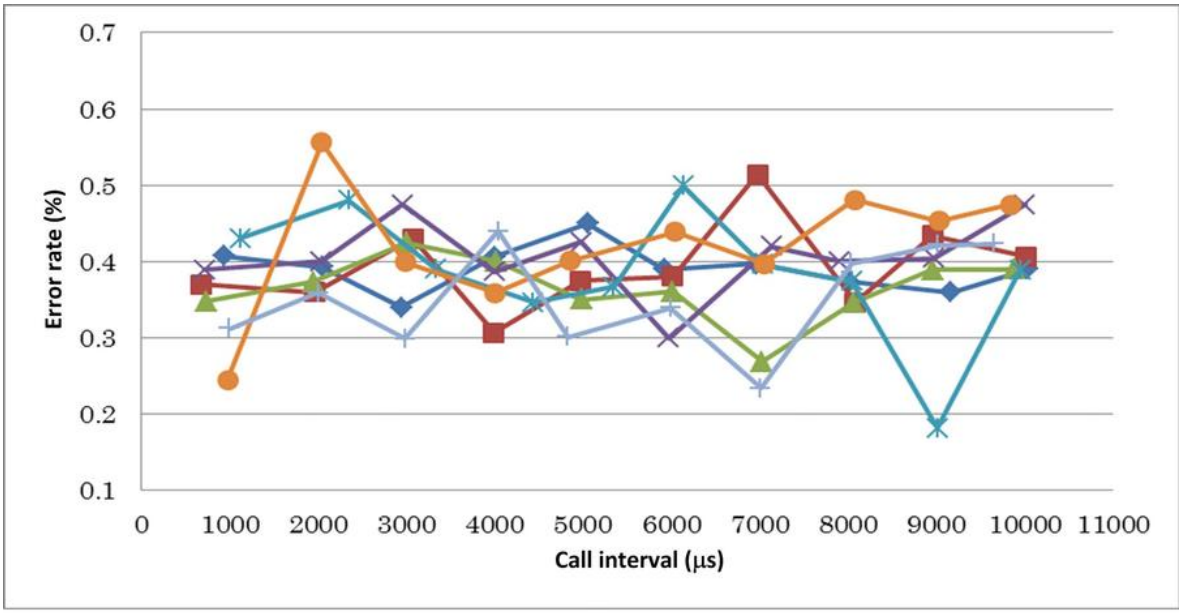
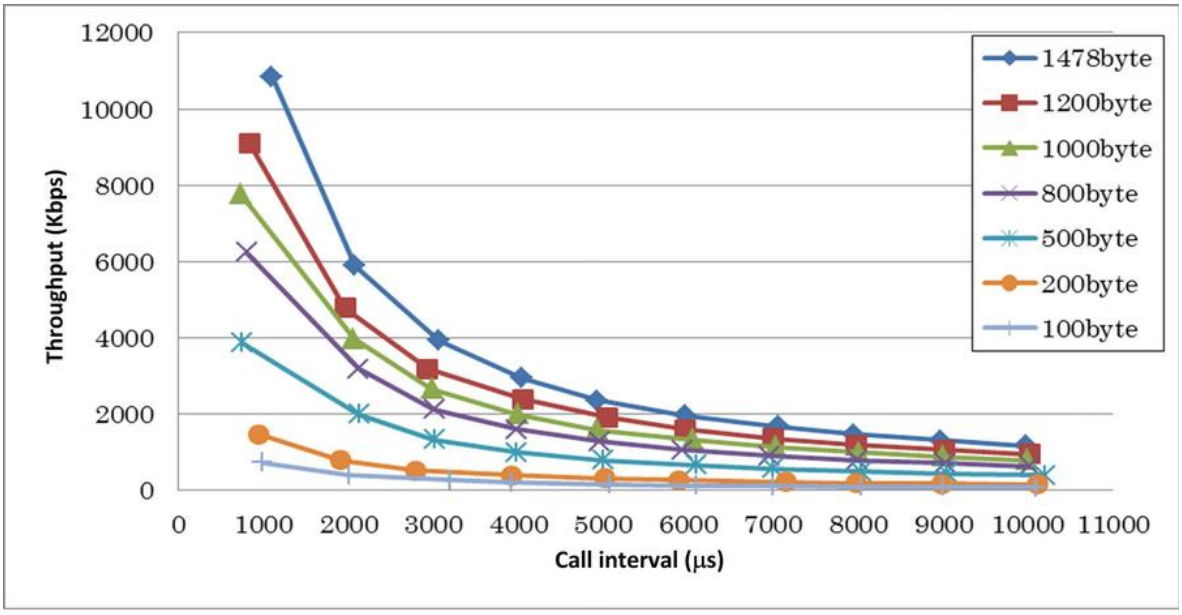
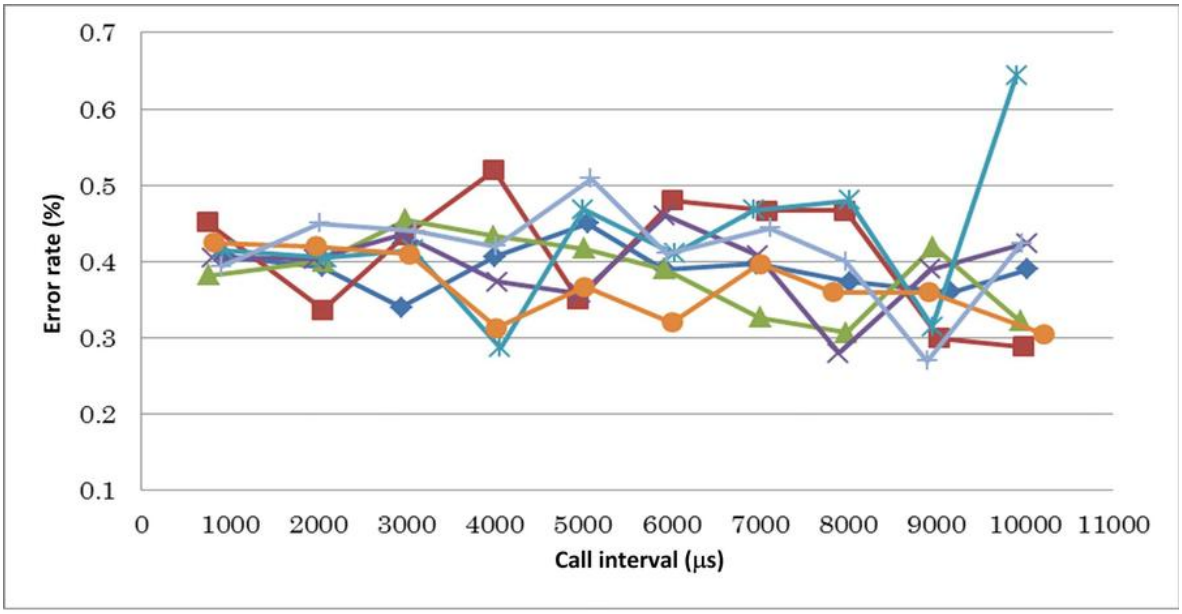




**Figure 7-4. Send Option = NO_WAIT | FORCE_DIRECT_BC**






# 8. Appendix: Cautions for Wireless Communication Environments

The 2.4 GHz wireless frequency bandwidth used by the 3DS for wireless communication continues to become an extremely crowded bandwidth, as it is used for a wide variety of wireless devices, first and foremost among them, Wi-Fi devices. In addition, worsening wireless conditions sometimes result in instability for 3DS wireless communication—a problem that cannot be solved with technology. Ideally, development would use wireless communication features and prepare conditions to remove interference

from other devices. The unique characteristics of wireless communication mean that it is not easy to create these conditions. In addition, a lack of understanding of wireless conditions can mistakenly lead people to believe that an environment is clear when it is not. This misunderstanding can add development time in many instances when trying to isolate the causes of a bug in communication.

This appendix describes several types of information related to interference between wireless devices, to further your understanding of wireless environments. It describes how to prepare a wireless environment free from the effects of other devices.

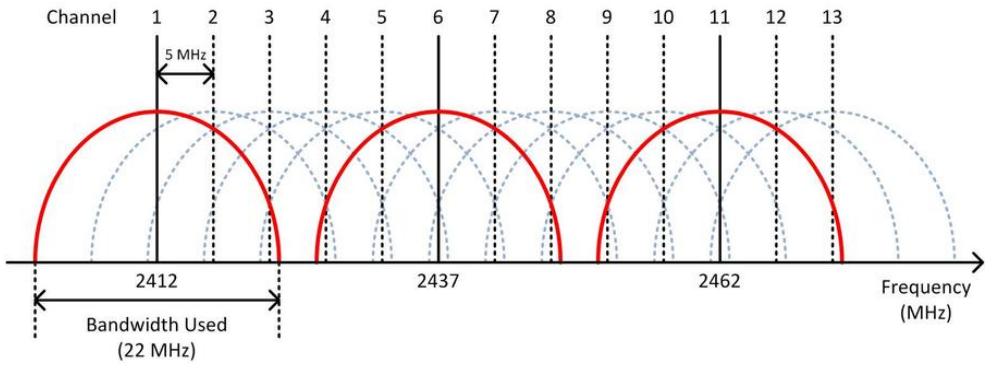
# 8.1. Cautions When Preparing a Wireless Communication Environment

The following section describes what you need to know to avoid interference between two wireless devices, and includes cautions to help you confirm whether you have set up a good wireless environment.

## 8.1.1. Setting Channels for Other Wi-Fi Devices

Figure 8-1 shows an example of assigning frequency channels in the 2.4 GHz bandwidth used by for 3DS wireless communication. As shown in the figure, 3DS wireless communication, and Wi-Fi communication in general, uses a wider amount of bandwidth than is assigned to a single channel. As a result, when devices are set with channels close to another, it results in interference in the communication of *both* devices. When you set channels to avoid interfering with other devices, we recommend a separation between devices of at least five channels.

**Figure 8-1. Assigning Channels in the 2.4 GHz Bandwidth**




The channels available for use differ by country.

However, even when devices are separated by at least five channels, it is impossible to completely eliminate mutual interference during communications. For example, even when two devices use channels 1 and 13 for communication, but are physically close to one another, they can both

interfere with each other. To eliminate the effect on both devices, set them to channels that are at least five channels apart and at a distance of at least two meters.

> **Note:** The necessary distance varies depending on amplification, signal strength, receive frequency, and device send output.

## 8.1.2. Interference From Devices Other Than Wi-Fi Devices

In addition to 3DS wireless communication and other general Wi-Fi communication under the IEEE 802.11 standard, the 2.4 GHz bandwidth is host to a wide variety of other communication methods and electronic devices. Such devices include microwave ovens, cordless phones, and Bluetooth devices (including peripherals for PCs and smartphones). There is no feature allowing for their efficient co-existence with IEEE 802.11, so interference from their signals can make communication unstable.

For a clear wireless environment, you must halt these other devices. Of particular concern is the smartphone, which now comes standard with Bluetooth. You must be cautious when bringing smartphones into the testing environment.

## 8.1.3. Caution Using Packet Capture

A wireless packet can be captured using packet capture tools available on the market, such as AirPcap (a wireless LAN protocol analyzer made by Riverbed Technology). While there are probably developers using packet capture tools to facilitate communication testing, the use of these tools does nothing to help determine how good or bad a wireless environment is, so take caution. In other words, even when the captured wireless packets are few in number or in an "empty state," communication may be unstable.

Packet capture tools do not capture packets using communication methods other than those set forth in IEEE 802.11. In addition, even when a wireless packet is IEEE 802.11-compliant, large portions of data might not be captured because they could not be decoded correctly due to weak signal reception. Note that even when a wireless packet is not captured, it can still be strong enough to have an effect on low-level processing such as the physical and MAC layers. In that case, such wireless packets indicate unstable communication.

> **Note:** When reception strength is weak, the protocol of the MAC layer sometimes does not operate correctly. A wireless packet that cannot be captured can have a more serious effect on communication than a wireless packet that can be captured.

# 8.2. Preparing a Wireless Communication Environment Unaffected by Other Devices

Given the intense pace at which wireless devices that use the 2.4 GHz bandwidth are spreading, it is likely to be difficult to yield a truly clear wireless environment, short of using an advanced electromagnetic black box or black room. Although it will not be completely clear, a comparatively clear wireless environment can be implemented through the use of commercially available electromagnetic shielding and coaxial cables.

> **Note:** Even with the wireless environment prepared using the methods introduced below, perform development and testing with the understanding that you cannot completely eliminate interference from other wireless devices. Being aware of the potential problems can help avoid unnecessary confusion and foster more efficient work.

## 8.2.1. Using an Electromagnetic Shield

One way to yield a clear wireless environment is to use a shield that blocks out electromagnetic field signals. There are a wide variety of shielding products on the market, ranging from the simple bag and box types to larger tent types. These products possess a degree of ability to shield electrical devices from external interference, so that more stable communication can be expected. However, take caution, as the majority of shielding products on the market reduce external interference, but cannot completely block it out.

As a general guideline, if the dampening capacity (attenuation capacity) of the shielding product is 40 dB or less, it cannot block, at a minimum, a wireless signal that is in the same room. To yield a clear wireless environment, use a shield with a dampening capacity of at least 60 dB, and avoid placing wireless devices near the shield.

> **Note:** Caution is needed because the shield can foster the harmful effects from dampened wireless packets.

**Figure 8-2. Various Electromagnetic Shields**

### 8.2.2. Using Coaxial Cables

With a debugger, devices can be connected by coaxial cable instead of the wireless antenna, and communication can occur using the coaxial cable. As with the electromagnetic shield, this has the benefit of controlling the effect of other wireless communication. However, because the signal that is exchanged via the cable is the same as the signal exchanged in wireless communication, there is no way to completely eliminate all external interference. When you use a coaxial cable, you must assume that there will be interference from wireless devices within a radius of a few meters.



# 9. Appendix: StreetPass Relay

StreetPass Relay is a mechanism similar to packet relay for passing StreetPass data among users connected to certain access points using the StreetPass Relay server. It creates opportunities for StreetPass to more users than just those who happen to be in the same place at the same time.

StreetPass Relay takes place for users connected to the following kinds of access points.

- Nintendo Zone
- Nintendo 3DS Station
- FREE SPOT (see **Note**)
- Wi-Fi (see **Note**)

  May not be available in some facilities.

**Note:** SteetPass Relay does not operate when the CTR system is connected to an access point registered in its Internet settings.

## 9.1. Differences From Normal StreetPass

The following sections describe ways in which the communications processes of StreetPass Relay differ from those of normal StreetPass.

**Note:** If StreetPass Relay will cause problems for your planned application, contact Nintendo to add the application to the list of unsupported titles.

## 9.1.1. All StreetPass Data Is a Target of Processing

With standard StreetPass, transmissions do not take place if the communication partners do not have StreetPass data for the same title. But with StreetPass Relay, all of the StreetPass data registered in a system is a subject of transmission. As a result, StreetPass communications take place once for each title for a maximum of 12 times.

If the StreetPass Relay server has StreetPass data for the same title that is in a 3DS system, the process is just like standard StreetPass, in that the StreetPass data is both sent and received. But if the StreetPass Relay server does not have StreetPass data for the same title, the only process that takes place is the sending (the uploading) of StreetPass data to the StreetPass Relay server.

## 9.1.2. Data Just for Friends Is Not Sent or Received

StreetPass Relay operates on the premise that StreetPass communications are taking place between users who are not friends. For this reason, if `MESSAGE_TYPEFLAG_FRIEND` is the only flag specified for the *`messageTypeFlag`* parameter when some StreetPass data is created, that data is not uploaded to the StreetPass Relay server.

Note that depending on your application's specifications, the StreetPass data that is passed between friends via direct StreetPass may differ from the StreetPass data that is passed via StreetPass Relay.

## 9.1.3. Exchange Peers Differ

Given the StreetPass Relay specifications, StreetPass data set with a transmission mode of Exchange is not necessarily exchanged among StreetPass users.

Assume that data has been exchanged between three users in the order A → B → C via StreetPass Relay. The process works as follows.

- User A sends data to the StreetPass Relay server. (StreetPass does not take place.)
- User B sends (user B) data to the StreetPass Relay server and receives user A's data from the server.
- User C sends (user C) data to the StreetPass Relay server and receives user B's data from the server.

In other words, user B receives user A's data, but user A does not receive user B's data.

## 9.2. Testing StreetPass Relay

Use the BossLotcheckTool to test StreetPass Relay with the development hardware. For more information, see the reference for the `BossLotcheckTool` tool in the CTR documentation. To test StreetPass Relay, you must prepare several units of the development hardware (a minimum of two, but three or more is recommended).

> **Note:** The development hardware will not perform the StreetPass Relay processes even if it is connected to one of the kinds of access points mentioned in 9. Appendix: StreetPass Relay. The StreetPass Relay server distinguishes between data received from retail devices and from development hardware, so that StreetPass communications do not take place between retail devices and development hardware.
>
> The StreetPass Relay test feature is supported by the BossLotcheckTool included in CRT-SDK.

The objective of testing StreetPass Relay is to determine whether your design can tolerate the changes in functionality that may arise due to the differences described in 9.1. Differences From Normal StreetPass. If you have implemented standard StreetPass correctly, there will be no problems or errors in the sending and receiving of StreetPass data, so you do not need to check whether StreetPass Relay can also do this normally.

## 9.3. Testing Procedure Using BossLotcheckTool

Follow this procedure to test StreetPass relay.

1. Register some StreetPass data.
2. Start BossLotcheckTool on the development hardware used in step (1), and from the menu select **Check functions** > **SPRelay** > **Forced Task Start**.
3. Wait for the message on the upper screen to change from "Executing SPRelay..."
4. Confirm that the changed message says "Executed(HttpStatus:200)"
   If another message displays, StreetPass Relay has not completed normally. For more information about the displayed messages, see 9.3.1. Execution Results of Forced Task Start.
5. Execute steps 1 through 4 on the other development hardware units.

> **Note:** Selecting **Forced Task Start** in the test procedure forces the StreetPass Relay process to run. The limitation of a maximum of 8 hours without transmissions between communication partners is ignored. The process runs via the access point that the development hardware

is connected to, so you can perform the testing without preparing a special communications environment like Nintendo Zone. The StreetPass Relay process runs even if no StreetPass data is registered in the CTR system or if the Parental Controls have been set to restrict the process. However, the execution result of **Forced Task Start** in an error.

When you conduct this test for the first time (when you connect to the StreetPass Relay server for the first time), only the sending of StreetPass data occurs. In subsequent tests using different development hardware, the StreetPass data is both sent and received, and when data is received the notification LED illuminates green, just like for regular StreetPass.

**Warning:** Do not select **Enable** in the menu that appears after you have selected **SPRelay**. Selecting this option sets the StreetPass Relay process to occur when the Nintendo Zone beacon is detected, which could hinder the testing of standard StreetPass.

## 9.3.1. Execution Results of Forced Task Start

The result of executing **Forced Task Start** is displayed in the following format.

```
$TaskResult(HttpStatus:$HttpStatusCode)
```

The $TaskResult portion shows the execution result of the BOSS process (the StreetPass Relay task) that was performed for StreetPass Relay. The table explains the execution that can be shown here.

**Table 9-1. StreetPass Relay Task Execution Results**

| Execution Results | Description of Execution Result |
|---|---|
| Executed | The StreetPass Relay task executed normally. The value 200 is always shown for the $HttpStatusCode portion of the result.<br>Even if the task executes normally, data sometimes is not exchanged for the following reasons.<br>• StreetPass Relay was performed consecutively using the same access point and the same development hardware. (If the uploaded StreetPass data is not passed to another system, the upload process itself will not work.)<br>• StreetPass Relay was run for the first time on the access point. (In this case, StreetPass data is only uploaded.) |
| Error | A nonretryable error occurred when the StreetPass Relay task executed.<br>If this execution result is displayed, check the HTTP status code displayed for $HttpStatusCode in addition to the following settings.<br>• Is StreetPass restricted in the Parental Controls settings?<br>• Is StreetPass data registered on the system? |

| | |
|---|---|
| Retry | A retryable error occurred when the StreetPass Relay task executed.<br>If this execution result displays, check the following settings.<br>• Is hardware connected to the network?<br>• Is the wireless signal weak?<br>• Was the system opened and closed or was the wireless switch manipulated during StreetPass Relay execution? |

The `$HttpStatusCode` portion of the execution result shows the HTTP status code returned from the StreetPass Relay server. The following table lists these HTTP status codes and explains their meanings.

**Table 9-2. HTTP Status Codes Returned From the StreetPass Relay Server**

| HTTP Status Code | Meaning of HTTP Status Code |
|---|---|
| 200 | StreetPass Relay succeeded. |
| 400 | StreetPass Relay failed.<br>There was a problem with the client request (development hardware setting). |
| 500 | StreetPass Relay failed.<br>There was an error on the StreetPass relay server. |
| 503 | StreetPass Relay failed.<br>The StreetPass relay server is down for maintenance. |

**Note:** The HTTP status codes returned by the StreetPass Relay server may change in the future.

# Revision History

## Version 1.6 (2016-06-24)

### Changes

- 3.3.3. Getting the Connection State
  - Added information about detecting connections to SpotPass hubs.

# Version 1.5 (2016-05-10)

## Changes

- 4.2.1.2. Setting Task Properties
    - Added information about conditions that reduce the execution count of the BOSS task.
- 4.2.2.1. NS Archives and NS Data
    - Added in the `SetOptoutFlag()` description that behavior is the same as when set to not receive notifications.
- 4.2.2.4. Preparing to Use BOSS
    - Added a warning about cases where registered tasks remain after unregistering BOSS storage with `nn::boss::UnregisterStorage()`.

# Version 1.4 (2015-11-05)

## Changes

- Overall
    - Added a definition name for the alignment restriction.
- 3.2.10. Emulated Clients
    - Added notes on releasing patches and revised versions.
- 4.1.1. Overview of Operations
    - Corrected notes describing situations where StreetPass communication does not occur.
- 4.1.3.1. Start Accessing (Opening and Creating)

    - Added notes describing the feature to get if a StreetPass box can be created.

- 4.1.4.1. Creating New Data
    - Added notes describing situations where either only send or receive succeeds.

- 4.2. SpotPass
    - Removed the section describing the Nintendo Zone region-specific data distribution.

- 4.2.1.3. Task Action Settings
    - Removed the section describing the Nintendo Zone region-specific data distribution.

- 4.2.2. Nintendo Archive Download (NADL) Tasks
    - Corrected the instructions for using the BOSS server to describe logging in to the BOSS data server and opening Help.

- 4.2.4. Data Upload Tasks
    - Corrected the information about reusing the data upload task.

- 4.2.5. Data Store Upload Tasks
    - Corrected the information about reusing the DataStore upload task.

- 6.1.1. Initializing
    - Added buffer size requirements.

- 6.1.2. Creating a Socket
    - Corrected information about the errors when creating a socket.

- 6.1.3. Binding Addresses and Port Numbers
    - Corrected information about the errors when binding addresses and port numbers.

- 6.1.4. Operating Modes
    - Corrected information about the errors when getting or setting a socket’s operating mode.

- 6.1.5. Waiting for Connections
    - Corrected information about the errors when waiting for connections.
- 6.1.6. Connecting to a Remote Host
    - Corrected information about the errors when checking connection to a remote host.
    - Corrected information about the errors when checking connection to a remote host.
- 6.1.7. Receiving Data
    - Corrected information about the errors when receiving data.
- 6.1.8. Sending Data
    - Corrected information about the errors when sending data.
- 6.1.9. Setting Options
    - Corrected information about the errors when setting options.
- 6.1.10. Disconnecting a Socket
    - Corrected information about the errors when disconnecting a socket.
- 6.1.11. Closing a Socket
    - Corrected information about the errors that occur when freeing a socket.
- 6.1.13. Utility Functions
    - Corrected information about the errors when obtaining a socket address.
    - Corrected information about the errors when getting host information with `nn::socket::GetAddrInfo()`.
    - Corrected information about the errors when getting host information with `nn::socket::GetNameInfo()`.
- 6.2.5. Handshaking
    - Added a warning that buffers must not be allocated in device memory.
- 6.3.1. Initializing

- Added a warning that buffers must not be allocated in device memory.

## Version 1.3 (2015-04-28)

### Additions

- 4.1.4.5. Updating Information
- 4.2.1.10. Handling Errors in the BOSS Library

### Changes

- 3.1.3.2. Creating a New Network
  - Fixed the hierarchy of the Config tools menu to match the current tools.
- 3.3. Automatic Connection
  - Fixed the hierarchy of the Config tools menu to match the current tools.
- 3.3.2. Connecting Automatically
  - Added a section about the order of connection priority when connecting automatically.
- 4.2.1.4. Registering and Running Tasks
  - Revised the text, changing "delete tasks" to "unregister tasks."
- 4.2.2.4. Preparing to Use BOSS
  - Added the section Maximum Number of BOSS Storage Regions That Can Be Registered and Automatic Unregistering.
- 4.2.2.5. Registering a Nintendo Archive Download (NADL) Task
  - Made changes to the description of checking the DNS server IP address.

- 4.2.2.8. Loading Downloaded Data
    - Revised the description of how the feature to prevent duplicate reception stops working correctly when tasks have been re-registered.
    - Added a description of how to determine whether NS data has been corrupted or tampered with.
- 5.4.4.1. Service Tokens for Independent Services
    - Added a jump to the Nintendo eShop patch page as a sequence to follow when the application needs to be updated.
- 6.1.7. Receiving Data
    - Corrected `MSG_PEEKN` to read `MSG_PEEK`.
    - Added that the maximum amount of data that can be received by a single function call is 1500 bytes if receiving with a datagram socket.
- 6.1.8. Sending Data
    - Added that the maximum amount of data that can be sent by a single function call is 1500 bytes if sending with a datagram socket.

## Version 1.2 (2015-01-15)

### Changes

- 4.2.2.2. BOSS Storage
    - Revised the note about Notifications.
- 4.2.2.8. Loading Downloaded Data
    - Revised the warning text about duplicate downloads.
- 6.1.7. Receiving Data
    - Revised the description of specifying a pointer to the `nn::socket::SockAddrIn` structure.

- 9.2. Testing StreetPass Relay
    - Added the reference target of BossLotcheckTool.

### Deletions

- Platform Notation
    - Deleted this page because the information about platform notation was moved to the Readme file.

## Version 1.1 (2014-11-10)

### Changes

- 4.2.3.2. Getting an NSA Listing
    - Added a description that the NSA list accesses the system language settings.

## Version 1.0 (2014-09-04)

### Additions and Changes

- Initial version.

# 3DS Programming Manual: Font Library

Version 1.1

**Nintendo Confidential**

This document contains confidential and proprietary information of Nintendo, and is protected under confidentiality agreements as well as the intellectual property laws of the United States and of other countries. No part of this document may be released, distributed, transmitted, or reproduced in any form , including by any electronic or mechanical means and by including within information storage and retrieval systems, without written permission from Nintendo.

©2015 Nintendo. All rights reserved.

All company and product names in this document are the trademarks or registered trademarks of their respective companies.

CTR-06-0147-002-C

# 1. Introduction

This document describes how to use the Font library, which is provided with the CTR-SDK, in programming procedures and other processes that render text characters in applications.

2. Terminology describes terminology specific to the Font library. Read it to learn about the terminology used in this document.

3. Font Management describes the classes used to manage font resources. Read it to learn about how font resources are managed in the Font library.

4. Displaying Characters describes methods of displaying text. Read it to learn about using the Font library to implement the displaying of text.

5. Creating Fonts describes methods of creating font resources using `ctr_FontConverter`. Read it if you want to create font resources to use in an application.

# 2. Terminology

This section defines terminology used with the Font library. The definitions used here are specific to this library and may differ from the ordinary meaning of the terms.

## 2.1. Terms Related to Characters and Text

The following table presents a summary of character-related and text-related terms.

**Table 2-1. Character-Related and Text-Related Terms**

| Term | Description |
|---|---|
| Letters | The graphical shape to render. It includes both the visually recognizable graphic image and the white space surrounding it. |
| Encoding | A numerical value assigned to each character to facilitate handling by computer systems. |
| Character Sets | A set of characters and their associated character codes. |
| Text Encoding | A method of converting or arranging character codes so that a string is represented as a byte array. |
| ISO 8859-1 | A character set formed by adding some extended Latin characters and symbols to the ASCII character set; also called Latin-1.<br>The range of character codes used is from `0x00` to `0xFF`, but `0x00` to `0x1F` and `0x80` to `0x9F` are allocated to control characters that cannot be displayed. |
| CP1252 (Code Page 1252) | An extended Latin character set defined by Microsoft for Windows. In this set, the characters in the range `0x80` to `0x9F`, which are allocated to control characters in ISO 8859-1, are replaced with displayable characters, so all of the displayable characters in ISO 8859-1 can also be displayed by CP1252. |
| JIS X 0201 | A Japanese character set defined in the JIS standards. It includes Japanese hiragana, katakana, and kanji as full-width characters. |
| JIS X 0208 | A Japanese character set defined in the JIS standards. It includes ASCII and half-width katakana as half-width characters. |
| Shift JIS | The standard character encoding in Japanese environments. The characters included in JIS X 0201 and JIS X 0208 can be used in this encoding. |
| Unicode | A character system intended to represent all characters used in the world in a single character set. It includes definitions for both character sets and character encodings. |
| UTF-16 | A character encoding defined under Unicode. In this encoding all characters are represented with two bytes, so it includes almost all other character encodings, but is not compatible with ASCII.<br>Depending on the endianness, it may use the notation UTF16-BE or UTF16-LE. |
| UTF-8 | A character encoding defined under Unicode. In this encoding the number of bytes corresponding to one character depends on the character. ASCII characters are represented with one byte, so this encoding is compatible with ASCII, in contrast to UTF-16. |

## 2.2. Terms Related to Glyphs

A *glyph* is defined to be a visually recognizable shape of a character. This section describes terms related to glyphs.

Figure 2-1. below illustrates parameters involved in the display of glyphs. The yellow regions are the glyphs, while the characters include both the glyphs and the surrounding green regions. Parameters of the glyphs that are supplied by the Font library are indicated in black, parameters of the fonts that are supplied by the Font library are indicated in blue, and parameters that are set by the application when displaying the glyphs as strings are indicated in red.

**Figure 2-1. Glyphs**

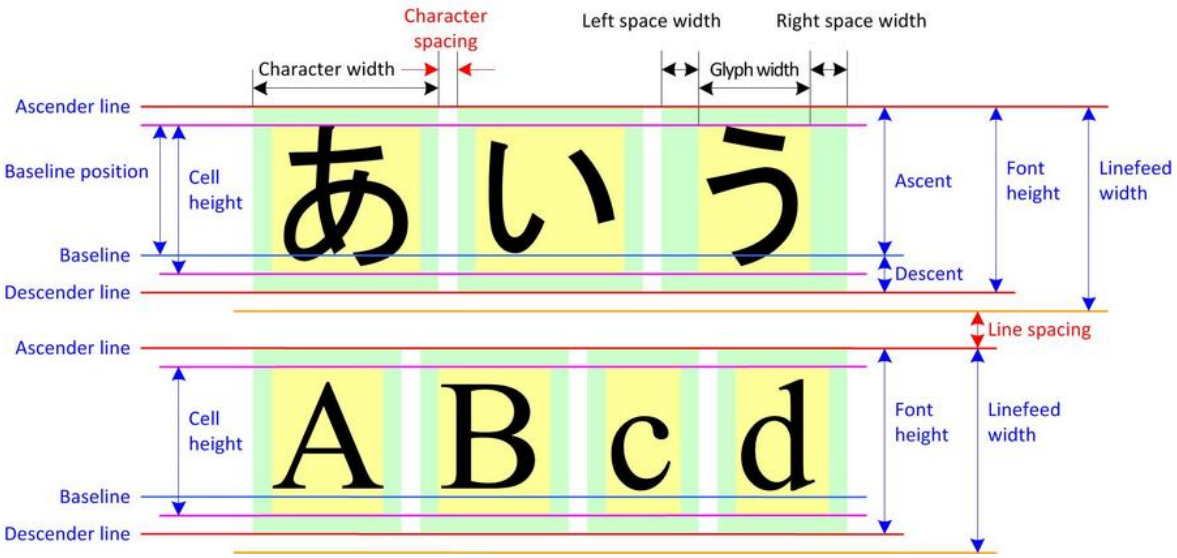



**Table 2-2. Glyph-Related Terms**

| Term | Description |
|---|---|
| Glyph Images | A term referring to a glyph that emphasizes its aspect of being a graphic image. It may also be called a *cell*. |
| Cell height | The height of a glyph image. This is common to the entire font. |
| Glyph width | The width of a rectangle circumscribing the glyph. In the figure, the width of the yellow region. |
| Left space width | The width of the space between the glyph and the immediately previous character. |
| Right space width | The width of the space between the glyph and the immediately subsequent character. |
| Character width | The width of the character. The sum of the glyph width and the left and right space widths. |

| | |
|---|---|
| Character width information | Information indicating the width of a character. Consists of the following four pieces of information: the left space width, the right space width, the glyph width and the character width. |
| Baseline | The position serving as the base in the vertical direction for all glyphs when lining up characters horizontally (that is, when rendering characters). |
| Ascender line | A line along the upper edge of the region to be occupied by the string. |
| Descender line | A line along the lower edge of the region to be occupied by the string. |
| Ascent | The distance from the baseline to the ascender line. |
| Descent | The distance from the baseline to the descender line. |
| Font height | The distance from the ascender line to the descender line. |
| Font width | The character width shared by all characters in a font. This width serves as the basis for the tab width. |

# 2.3. Terms Related to Font Resources

The following table presents terms related to font data and other font resources.

**Table 2-3. Terms Related to Font Resources**

| Term | Description |
|---|---|
| BCFNT (Binary Ctr FoNT) | Ordinary font data used in the Font library. Also, the extension for files that contain this font data. |
| BCFNA (Binary Ctr FoNt Archived) | Compressed font data used in the Font library. Also, the extension for files that contain this font data.<br>You can extract and decompress individual glyph sets as needed from these archives. . These archives are used when the fonts to be displayed are grouped by language, or when you want to switch only some fonts to a different typeface. |
| Font resource | Ordinary font data used in the Font library. Synonymous with BCFNT. |
| Archived font | Compressed font data used in the Font library. Synonymous with BCFNTA. |
| `ctr_FontConverter` | A Windows tool used to create BCFNT and BCFNA data. Both a GUI version and command-line version are provided. |
| Alternate character | A substitute character used when information for the requested character is not contained within a font. |

## 2.3.1. Glyph Groups

A *glyph group* is defined as a glyph set that is subdivided into groups. In addition, a set of glyph groups is called a *group set*.

The Font library provides a feature for constructing a font by freely selecting and combining any of the groups within a group set. For example, from a group set comprising the four groups "ascii," "latin," "greek," and "cyrillic," it is possible to construct a font containing only the glyphs required to display text for languages that use the Latin alphabet by combining "ascii" and "latin," or a font for languages that use the Cyrillic alphabet by combining "ascii" and "cyrillic."

**Figure 2-2. Examples of Glyph Group**

### 2.3.2. Sheets

A *sheet* is defined as a texture object upon which glyph images are drawn. One texture object corresponds to one sheet. Multiple glyph images can be arranged on a single sheet.

**Figure 2-3. Examples of Sheets**

## 2.4. Other Terms

The following table presents other terms related to the Font library.

**Table 2-4. Other Terms**

| Term | Description |
|---|---|
| Tagged strings | Strings that contain instructions related to rendering text.<br>The Font library provides a framework for processing tagged strings. The actual processing of tags embedded in strings must be handled by the application. |

CONFIDENTIAL

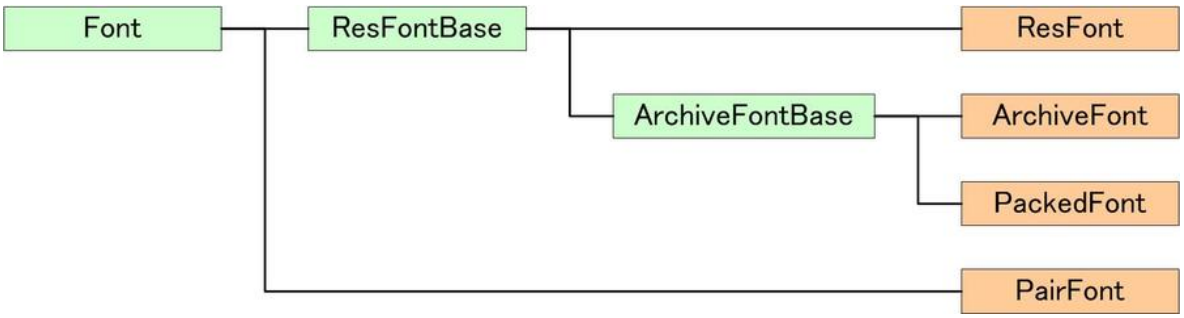
# 3. Font Management

This chapter describes the management of fonts. The rendering of characters by applications is described in subsequent chapters.

The `Font` library provides the `Font` class as a base class, the `ResFont` class for manipulating font resources loaded into memory, the `ArchiveFont` class and `PackedFont` class for manipulating archived font resources, and the `PairFont` class for handling two fonts as a single font. Note that we recommend using UTF16-LE as the encoding for font resources, because this is the encoding used by other CTR-SDK libraries to handle strings.

The following figure illustrates the hierarchy of the Font library classes (with derived classes on the right). In your application, select from among the classes along the right edge of the figure and create instances of these classes as necessary depending on how you use font resources.

**Figure 3-1. Hierarchy of Classes (for Font Management)**




## 3.1. Font Class

This is the base class for handling font resources, but this class itself is never instantiated. When the Font library classes are used to render characters, textures (sheets upon which glyph images are drawn) are loaded by directly referencing main memory, so the application must prepare a buffer in device memory for the font (the images). The restrictions placed on the location of this memory are described in the individual classes that inherit from this class.

Table 3-1 is a list of the member functions defined in the `Font` class. Note that as a general rule, when these functions get parameters they return values in pixels.

**Table 3-1. List of Member Functions of the Font Class**

| Category | Function | Description |
|---|---|---|
| Get font information | `GetWidth` | Returns the font width. Used to determine the tab width and perform similar tasks. |
| | `GetHeight` | Returns the font height. |
| | `GetAscent` | Returns the ascent. Used to align the upper edges of characters. |
| | `GetDescent` | Returns the descent. Used to align the lower edges of characters. |
| | `GetMaxCharWidth` | Returns the maximum character width of all of the characters contained in the font. |
| | `GetType` | Returns the font type. |
| | `GetTextureFormat` | Returns the format of the sheet (the texture). |
| | `GetLineFeed` | Returns the linefeed width. |
| | `GetDefaultCharWidths` | Returns the default character width information. Used as the character width for characters not contained in the font and for other anomalous widths. |
| Set font information | `SetLineFeed` | Resets the linefeed width. |
| | `SetDefaultCharWidths` | Resets the default character width information. |
| | `SetAlternateChar` | Sets alternate characters. |
| Get character information | `GetCharWidth` | Returns the character width of a character. |
| | `GetCharWidths` | Returns the character width information of a character. |
| | `GetGlyph` | Glyph information. Returns the glyph of an alternate character if the glyph of the specified character is not contained in the font. |
| | `HasGlyph` | Returns whether the glyph of the specified character is contained in the font. |
| String Encoding | `GetCharacterCode` | Returns the character encoding. |
| | `GetCharStrmReader` | The `CharWriter` class is used when drawing characters with the Font library. |
| Gets sheet information. | `GetBaselinePos` | Returns the baseline position. |
| | `GetCellHeight` | Returns the cell height. |
| | `GetCellWidth` | Returns the cell width. |

| | | |
|---|---|---|
| Texture interpolation | `EnableLinearFilter` | When rendering characters with the Font library, sets whether to use linear interpolation when scaling characters. |
| | `IsLinearFilterEnableAtSmall` | When rendering characters with the Font library, returns whether linear interpolation is used when scaling down characters. |
| | `IsLinearFilterEnableAtLarge` | When rendering characters with the Font library, returns whether linear interpolation is used when scaling up characters. |
| | `GetTextureWrapFilterValue` | When rendering characters with the Font library, returns the wrapping mode and filter settings. |

## 3.2. ResFont Class

This is a class for manipulating font resources. It is not compatible with grouping, so when using this class all sheets are loaded in memory. For this reason, the entire font resource must be loaded in memory.

Font resources (BCFNT files) loaded in memory are bound to the `ResFont` class by the `SetResource()` function. When the Font library classes are used to render characters, the `glGenTextures()` function is used to create one texture object for each sheet contained in the font resource. These created texture objects are configured so that the GPU directly accesses their texture images. Accordingly, the font resource must be loaded into device memory using 128-byte alignment (defined by `nn::font::GlyphDataAlignment`). Unbind the resource using the `RemoveResource()` function. Do not release the buffer into which the font resource was loaded until the resource is unbound.

When the Font library classes are used to render characters, a render buffer having the size returned by the `GetDrawBufferSize()` function must be set using the `SetDrawBuffer()` function, but there is no need to allocate the render buffer in device memory. The render buffer must be aligned using the `nn::font::ResFont::BUFFER_ALIGNMENT` (4 byte) value.

### 3.2.1. System Fonts

With 3DS, font resources called *system fonts* are stored in system memory. The application can use the PL library to use these system fonts as shared font resources.

Calling the `nn::pl::InitializeSharedFont()` function starts the loading of the system fonts corresponding to the 3DS system region, and the loading is complete when the `nn::pl::GetSharedFontLoadState()` function returns `nn::pl::SHARED_FONT_LOAD_STATE_LOADED`. Font resources loaded by this procedure are loaded to fixed addresses (physical addresses outside of application jurisdiction), so they do not consume any application memory.

The `nn::pl::GetSharedFontType()` function gets the type of the system fonts, which are loaded based on the 3DS system's region. To use a system font of a type not returned by this function, you must use the `nn::pl::MountSharedFont()` function to mount the system font archive and then have the application load the font resource file. The files are compressed in LZ77 format, so they must be uncompressed using the CX Library. Also note that each type of system font must be mounted separately, and that the uncompressed font resources must be placed in device memory to render characters with the Font library. When the loading of font resources is complete, unmount the archive with the `nn::pl::UnmountSharedFont()` function.

**Note:** For more information about loading system fonts, see the *3DS Programming Manual: System*.

## 3.3. ArchiveFont Class

This is a class for manipulating archived fonts. The glyphs within the font files are organized into groups, so you can uncompress and load into memory sheets containing just the specified groups. (Multiple groups can be specified.) To specify groups, either use `LOAD_GLYPH_ALL`, which loads all glyphs, or use a string containing the names of the groups to load delimited by commas (",").

There are two ways to construct fonts: (1) you can construct fonts from archived font data that is already loaded in memory, or (2) you can load the data into memory as a stream to reduce the amount of memory required for loading. Use the `Construct()` function for the former method and the `InitStreamingConstruct()` and `StreamingConstruct()` functions for the latter. Font data is compressed, but the font images are decompressed and loaded into memory. Use the `GetRequireBufferSize()` function to get the size of the buffer required to construct the font. (The data is decompressed into this buffer.) When the Font library classes are used to render characters, the buffer must be allocated in device memory using 128-byte alignment.

After the font is constructed, the memory used to load archived fonts can be released, but the buffer used to construct the font cannot be released until the font is destroyed. Destroy the font using the `Destroy()` function. When this function destroys a font, it returns a pointer to the buffer for you to then release.

## 3.4. PackedFont Class

Like the `ArchiveFont` class, the `PackedFont` class is a class for manipulating archived fonts. However, by using this class, you can maintain font resources in memory in compressed form and decompress and use them as needed. Although decompressing the sheet corresponding to a particular character entails a processing load, this class implements a cache so that the load is

reduced somewhat. By managing the proportion of data left in the cache and locking the sheets, you can prevent cache misses.

The `PackedFont` class uses the same font construction methods as the `ArchiveFont` class. When calculating the size of the buffer, specify the number or proportion of the sheets to be left in the cache. The following functions are provided for managing the cache: `GetNumCache` which gets the number of cacheable sheets, `PreloadSheet` which preloads the cache with sheets, `LockSheet` which locks sheets, `UnlockSheet` which unlocks individual sheets and `UnlockSheetAll` which unlocks all sheets.

After the font is constructed, the memory used to load archived fonts can be released, but the buffer used to construct the font cannot be released until the font is destroyed. Destroy the font using the `Destroy()` function. When this function destroys a font, it returns a pointer to the buffer for you to then release.

## 3.5. PairFont Class

This class inherits the `Font` class and enables you to use two fonts that have the same character encoding and same texture format as a single font. Note that problems with the display positions of characters might occur when combining fonts that have different baseline positions.

When an instance of the `PairFont` class is constructed, the font specified first is called the *primary font* and the font specified next is the *secondary font*. When you get parameters using this class, the class returns parameters for either the primary font or secondary font in accordance with the conditions in the following table.

**Table 3-2. Parameters of the PairFont Class**

| Parameter | Parameter Returned |
|---|---|
| Font width | Greatest value among the two fonts. |
| Font height | |
| Maximum character width | |
| Ascent | The value for the font that has the largest font height.<br>If both fonts have the same height, the value for the primary font is returned. |
| Descent | |
| Baseline | |
| Cell height | |
| Line feed width | |
| Cell width | |

| | |
|---|---|
| Default character width | The value for the font that has the largest font width.<br>If both fonts have the same width, the value for the primary font is returned. |
| Getting glyphs | The primary font takes priority.<br>The glyph from the secondary font is returned if the specified character is not present in the primary font. The alternate character from the primary font is returned if the specified character is not in either font. However, if an alternate character is set after the instance was constructed, and that alternate character is not present in the primary font but is present in the secondary font, the alternate character from the secondary font is returned. |
| Alternate character | The primary font takes priority.<br>However, if an alternate character is set after the instance was constructed, and that alternate character is not present in the primary font but is present in the secondary font, the alternate character from the secondary font is returned. |

CONFIDENTIAL

# 4. Displaying Characters

There are two ways to display characters: by using glyph information obtained from the font resources, or by using the character rendering classes provided by the Font library.

## 4.1. Displaying Characters Based on Glyph Information

The `GetGlyph()` function of the `Font` class returns glyph information in a `Glyph` structure. This structure contains the information required to display glyphs as characters and is defined as follows.

**Code 4-1. Glyph Structure**

```
struct Glyph
{
    const void* pTexture;
    struct CharWidths
    {
        s8  left;
        u8  glyphWidth;
        s8  charWidth;
    } widths;
    u8      height;
    u16     texWidth;
    u16     texHeight;
    u16     cellX;
    u16     cellY;
    u8      isSheetUpdated;
    TexFmt  texFormat;
    const TextureObject* pTextureObject;
};
```

The values stored in the various members are as follows. The values of the widths and heights are stored in pixel units.

**Table 4-1. Members of the Glyph Structure**

| Member | Description |
|---|---|
| pTexture | Pointer to the sheet (the texture image). |
| widths.left | Left space width. |
| widths.glyphWidth | Glyph width. |
| widths.charWidth | Character width.<br>The right space width can be found by this formula: [character width – glyph width – left space width]. |
| height | Cell (glyph image) height. |
| texWidth | Sheet width. |
| texHeight | Sheet height. |
| cellX | X-coordinate of the upper-left corner of the cell, taking the upper-left corner of the sheet as the origin. |
| cellY | Y-coordinate of the upper-left corner of the cell, taking the upper-left corner of the sheet as the origin. |
| isSheetUpdated | Sheet update flag. Referenced by the Font library. |
| texFormat | Sheet (texture) format. |
| pTextureObject | Texture object information and other information. Referenced by the Font library. |

The sheet format definitions correspond as follows to the argument values specified to the glTexImage2D() function.

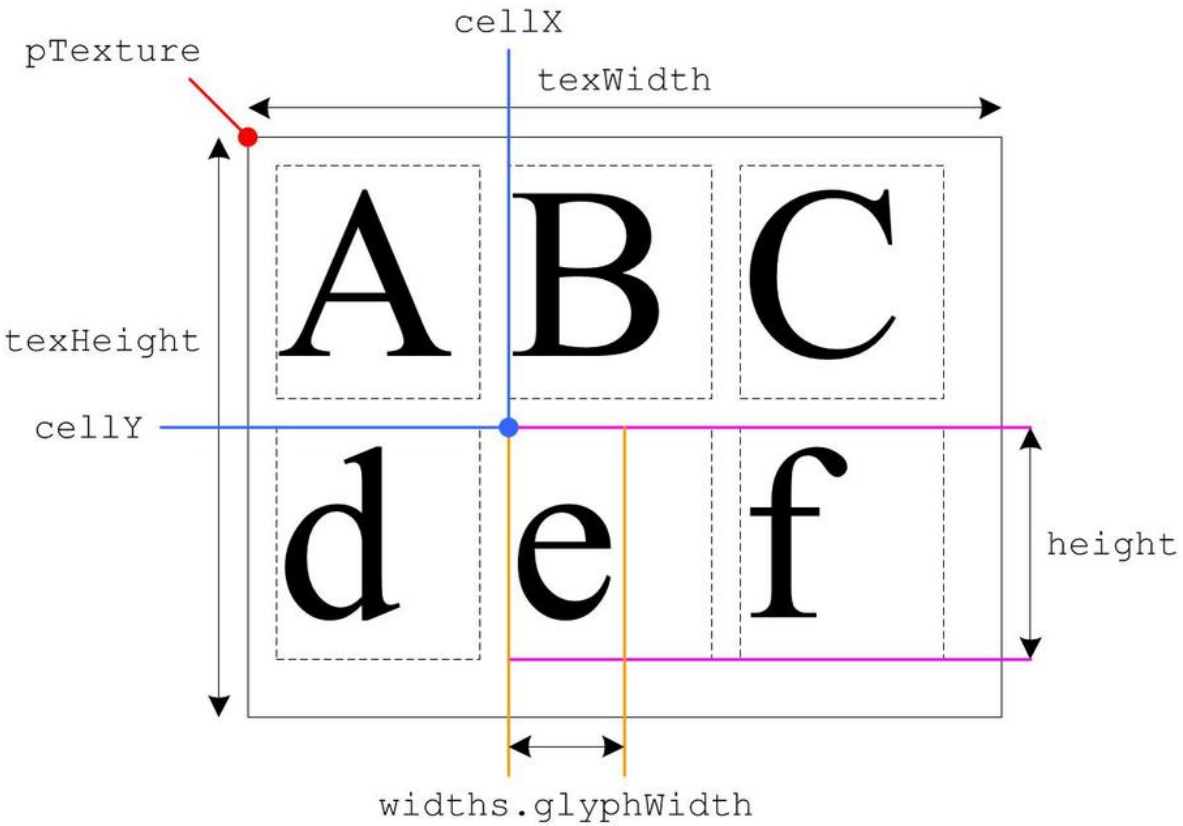
**Table 4-2. Sheet Format Definitions and Argument Values Specified to the glTexImage2D Function**

| Format | format | type |
|---|---|---|
| FONT_SHEET_FORMAT_A4 | GL_ALPHA_NATIVE_DMP | GL_UNSIGNED_4BITS_DMP |
| FONT_SHEET_FORMAT_A8 | GL_ALPHA_NATIVE_DMP | GL_UNSIGNED_BYTE |
| FONT_SHEET_FORMAT_LA4 | GL_LUMINANCE_ALPHA_NATIVE_DMP | GL_UNSIGNED_BYTE_4_4_DMP |
| FONT_SHEET_FORMAT_LA8 | GL_LUMINANCE_ALPHA_NATIVE_DMP | GL_UNSIGNED_BYTE |
| FONT_SHEET_FORMAT_RGBA4 | GL_RGBA_NATIVE_DMP | GL_UNSIGNED_SHORT_4_4_4_4 |
| FONT_SHEET_FORMAT_RGB5A1 | GL_RGBA_NATIVE_DMP | GL_UNSIGNED_SHORT_5_5_5_1 |
| FONT_SHEET_FORMAT_RGBA8 | GL_RGBA_NATIVE_DMP | GL_UNSIGNED_BYTE |
| FONT_SHEET_FORMAT_RGB565 | GL_RGB_NATIVE_DMP | GL_UNSIGNED_SHORT_5_6_5 |
| FONT_SHEET_FORMAT_RGB8 | GL_RGB_NATIVE_DMP | GL_UNSIGNED_BYTE |

The following structure members are required in order to identify the location in the sheet of the glyph

image to display.

**Figure 4-1. Glyph Structure Members Required to Identify the Location of a Glyph Image**




You must note that while the 3DS texture coordinates have their origin at the lower left, the origin of the cell (glyph image) coordinates is at the upper left. Because of this, the calculation of texture coordinates is implemented as in the following sample code.

**Code 4-2. Calculation of Texture Coordinates**

```
Glyph glyph;
myFont.GetGlyph(&glyph, character);

const f32 texLeft   = 1.0f * glyph.cellX / glyph.texWidth;
const f32 texRight  = 1.0f * (glyph.cellX + glyph.widths.glyphWidth)
                    / glyph.texWidth;
// Be careful about the differences between glyph image coordinates and texture
coordinate origins.
const f32 texTop    = 1.0f * (glyph.texHeight - glyph.cellY) / glyph.texHeight;
const f32 texBottom = 1.0f * (glyph.texHeight - (glyph.cellY + glyph.height))
                    / glyph.texHeight;
```

## 4.2. Using the Font Library Classes

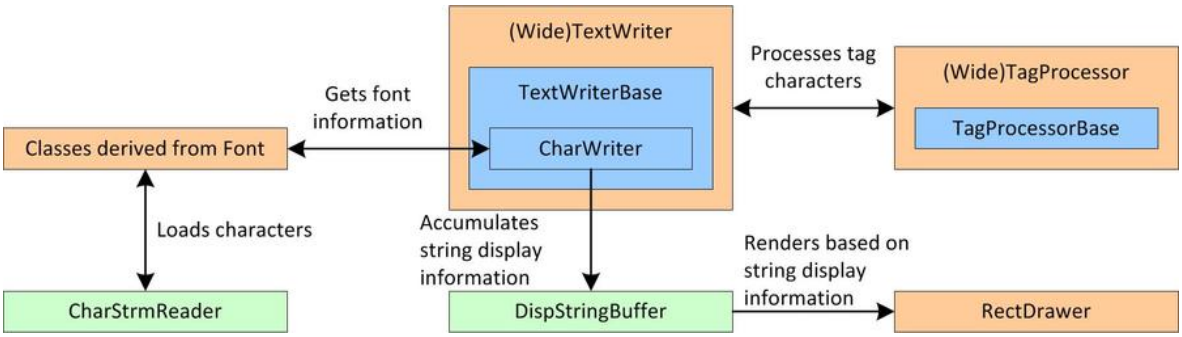
The Font library provides classes for rendering characters and strings. In addition to simply rendering

single characters, it also has the following features:

- Changing the text color
- Vertical or horizontal color gradation on the per-character level
- Scaling characters
- Forced fixed-width (monospaced) font display
- Character spacing, line spacing, and tab spacing
- Left-justification or right-justification
- Automatic line wrapping (newlines) in text
- Rendering using format strings (`printf` equivalent)
- Processing of tagged strings (customizable)

Several classes are used to render characters with the Font library, and they cooperate as illustrated below. In this process flow, the classes denoted by orange rectangles in the figure below are classes that configure settings for rendering text, and they are instantiated by the application. Also, blue denotes base classes, while green denotes implicitly used classes.

**Figure 4-2. Font Library Class Structure (for Text Rendering)**




If you use Font library classes to render text, we recommend using the RectDrawer class, which directly generates 3D commands (PICA register settings).

**Table 4-3. Processing Performed by Classes Used in Rendering**

| Class | Processing Performed |
|---|---|
| CharWriter | Renders individual characters. If you do not use `TextWriter`, inherit this class and form the characters into strings, along with the other processing involved with rendering strings. |
| TextWriter | Renders strings. This class provides the basic features for generating strings. |
| TagProcessor | Processes tag characters (control characters in the range `0x0000` to `0x001F`). If you want the tag processor to support characters other than the newline character (`0x000A`) and the tab character (`0x0009`), define a custom class that inherits `TagProcessor`. |
| RectDrawer | Generates 3D commands from the string display information accumulated in a buffer by the `TextWriter` (or `CharWriter`) class. Generates 3D commands compatible with dedicated shader programs. |

## 4.2.1. Steps for Implementing in an Application

This section describes the procedures for using the Font library classes to display strings on the screen, using as an example the source code for the `ResFont` demo included in the sample demos.

### 4.2.1.1. Loading and Constructing Font Resources

First, load and construct the font resources used to render text. In the sample demo, a pointer to an instance of the `ResFont` class and the font resource filename (that has an extension of `bcfnt`) are passed to the `InitFont()` function.

**Code 4-3. Loading and Constructing Font Resources**

```
//------------------------------------------------------------------------
//! @brief      Constructs ResFont.
//!
//! @param[out] pFont:    Pointer to the font to construct.
//! @param[in]  filePath: Filename of the font resource to load.
//!
//! @return     Returns whether ResFont was successfully constructed.
//------------------------------------------------------------------------
bool
InitFont(
    nn::font::ResFont*  pFont,
    const char*         filePath)
{
    // Load font resources.
    nn::fs::FileReader fontReader(filePath);

    s32 fileSize = (s32)fontReader.GetSize();
    if ( fileSize <= 0 )
    {
        return false;
    }

    void* buffer = s_AppHeap.Allocate(fileSize, nn::font::GlyphDataAlignment);
    if (buffer == NULL)
    {
        return false;
    }

    s32 readSize = fontReader.Read(buffer, fileSize);
    if (readSize != fileSize)
    {
        s_AppHeap.Free(buffer);
        return false;
    }

    // Set font resources.
    bool bSuccess = pFont->SetResource(buffer);
    NN_ASSERT(bSuccess);

    //---Fails when resources are already set or already loaded,
    //    or when the resources are invalid.
    if (! bSuccess)
    {
        s_AppHeap.Free(buffer);
    }
```

```
    // Set the render buffer.
    const u32 drawBufferSize = nn::font::ResFont::GetDrawBufferSize(buffer);
    void* drawBuffer = s_AppHeap.Allocate(drawBufferSize, 4);
    NN_NULL_ASSERT(drawBuffer);
    pFont->SetDrawBuffer(drawBuffer);

    return bSuccess;
}
```

In the Font library, textures are loaded with settings such that the GPU directly accesses the sheets that contain the drawn glyph images of the loaded font resources. Accordingly, font resources must be loaded into a buffer in device memory that has 128-byte alignment. Note that the only restriction on the render buffer is the requirement of 4-byte alignment, so this buffer does not need to be allocated in device memory.

For more information about the restrictions that apply when using font resources other than the ResFont class, see 3.3. The ArchiveFont Class and 3.4. The PackedFont Class.

## 4.2.1.2. Initializing Shaders

Load the shader binary that will be used for text display, and allocate a buffer for the `RectDrawer` class. In the sample demo, a pointer to an instance of the `RectDrawer` class is passed to the `InitShaders()` function.

**Code 4-4. Initializing the Shaders**

```
//------------------------------------------------------------------------
//!  @brief      Initializes the shaders.
//!
//! @param[in,out]  pDrawer    Pointer to the instance to initialize.
//------------------------------------------------------------------------
void*
InitShaders(nn::font::RectDrawer* pDrawer)
{
    nn::fs::FileReader shaderReader(s_ShaderBinaryFilePath);

    const u32 fileSize = (u32)shaderReader.GetSize();

    void* shaderBinary = s_AppHeap.Allocate(fileSize);
    NN_NULL_ASSERT(shaderBinary);

#ifndef NN_BUILD_RELEASE
    s32 read =
#endif // NN_BUILD_RELEASE
    shaderReader.Read(shaderBinary, fileSize);
    NN_ASSERT(read == fileSize);

    const u32 vtxBufCmdBufSize =
        nn::font::RectDrawer::GetVertexBufferCommandBufferSize(
            shaderBinary, fileSize);
    void *const vtxBufCmdBuf = s_AppHeap.Allocate(vtxBufCmdBufSize);
    NN_NULL_ASSERT(vtxBufCmdBuf);
    pDrawer->Initialize(vtxBufCmdBuf, shaderBinary, fileSize);

    s_AppHeap.Free(shaderBinary);

    return vtxBufCmdBuf;
}
```

In the `RectDrawer` class, two buffers are used: the vertex buffer and the command buffer. The vertex buffer contains vertex coordinates for use in rendering. The command buffer stores render commands, but also contains shader binary loading commands (programs, swizzle patterns, constant registers) and other such commands. These buffers are accessed directly by the GPU, so they must be allocated in device memory. There is no required buffer alignment, but we recommend 4-byte or 16-byte alignment.

In the sample demo, the two buffers are allocated in a single region, but functions for getting the required size of each buffer and an initialization function for allocating the buffers separately are also provided. After initialization is successful, you can release the buffer that contains the shader binary.

The SDK contains `nnfont_RectDrawerShader.shbin` as the shader binary used in the `RectDrawer` class. This shader assumes that an orthogonal projection matrix has been set in which the sizes of the projected screen in screen space and the LCD screen are identical, with the origin of the projection matrix at the upper left of the screen in screen space (although the directions of the y-axis and z-axis are the reverse of the 3DS axes), and also assumes that the modelview matrix is set to a unit matrix. The adjustment of the character display position and size is managed by accumulating transformation matrices in the floating-point constant registers.

## 4.2.1.3. Allocating a Display String Buffer

Allocate a buffer that accumulates string display information (the display string buffer). The required size of the display string buffer is calculated based on the maximum number of characters to display. When string rendering is complete, the same buffer can be used to accumulate string display information. If the font and characters to display do not change, the stored string display information can be reused unchanged. If the string has previously been rendered in a single color, you can change just the font color and do not need to accumulate the string display information again.

**Code 4-5. Allocating a Display String Buffer**

```
//------------------------------------------------------------------------
//!  @brief      Allocates a display string buffer.
//!
//!  @param[in]  charMax: Maximum number of characters in the string to
display.
//!
//!  @return     Returns a pointer to the allocated display string buffer.
//------------------------------------------------------------------------
nn::font::DispStringBuffer*
AllocDispStringBuffer(int charMax)
{
    const u32 DrawBufferSize =
        nn::font::CharWriter::GetDispStringBufferSize(charMax);
    void *const bufMem = s_AppHeap.Allocate(DrawBufferSize);
    NN_NULL_ASSERT(bufMem);

    return nn::font::CharWriter::InitDispStringBuffer(bufMem, charMax);
}
```

There is no need to allocate the display string buffer in device memory. There is no required buffer alignment, but we recommend 4-byte alignment.

## 4.2.1.4. Rendering Settings

The Font library configures the minimum level of settings required for rendering characters, so the application must appropriately set the following render pipeline stages before rendering.

- Culling
- Scissor test
- Polygon offset
- Early depth test
- Depth test
- Stencil test
- Masking
- Framebuffer object

If the application makes culling or depth test settings to display 3D models or the like, the settings in effect immediately before the text is rendered are applied to the text. However, you can change these settings before rendering.
In the following sample demo, the `InitDraw()` function generates commands that configure the rendering settings.

**Code 4-6. Rendering Settings**

```
//------------------------------------------------------------------------
//!  @brief      Performs the initial settings for rendering.
//!
//! @param[in]  width             Screen width.
//! @param[in]  height            Screen height.
//------------------------------------------------------------------------
void
InitDraw(
    int width,
    int height
)
{
    // Color buffer information.
    // Align with the orientation of the LCD, and swap the width with the
height.
    const nn::font::ColorBufferInfo colBufInfo =
    { width, height, PICA_DATA_DEPTH24_STENCIL8_EXT };

    const u32 screenSettingCommands[] =
    {
        // Viewport settings.
        NN_FONT_CMD_SET_VIEWPORT( 0, 0, colBufInfo.width, colBufInfo.height ),

        // Disable scissoring.
        NN_FONT_CMD_SET_DISABLE_SCISSOR( colBufInfo ),
```

```
            // Disable the w buffer.
            // Depth range settings.
            // Disable polygon offset.
            NN_FONT_CMD_SET_WBUFFER_DEPTHRANGE_POLYGONOFFSET(
                0.0f,   // wScale : 0.0 disables the W buffer.
                0.0f,   // Depth range near.
                1.0f,   // Depth range far.
                0,      // Polygon offset units : 0.0 disables the polygon offset.
                colBufInfo),
        };
        nngxAdd3DCommand(screenSettingCommands, sizeof(screenSettingCommands),
    true);

        static const u32 s_InitCommands[] =
        {
            // Disable culling.
            NN_FONT_CMD_SET_CULL_FACE( NN_FONT_CMD_CULL_FACE_DISABLE ),

            // Disable stencil test.
            NN_FONT_CMD_SET_DISABLE_STENCIL_TEST(),

            // Disable depth test.
            // Enable writing of all color buffer components.
            NN_FONT_CMD_SET_DEPTH_FUNC_COLOR_MASK(
                false,  // isDepthTestEnabled
                0,      // depthFunc
                true,   // depthMask
                true,   // red
                true,   // green
                true,   // blue
                true),  // alpha

            // Disable early depth test.
            NN_FONT_CMD_SET_ENABLE_EARLY_DEPTH_TEST( false ),

            // Control access to the framebuffer.
            NN_FONT_CMD_SET_FBACCESS(
                true,   // colorRead
                true,   // colorWrite
                false,  // depthRead
                false,  // depthWrite
                false,  // stencilRead
                false), // stencilWrite
        };
        nngxAdd3DCommand(s_InitCommands, sizeof(s_InitCommands), true);
    }
```

## 4.2.1.5. Processing Each Time Text Is Rendered

The processing performed until now is just initialization. This section describes the processing performed each time text is rendered.

The `(Wide)TextWriter` classes perform the rendering of strings that have a specified display position and font color. An instance of this class is assumed to be generated each frame, so this class has no explicit initialization function. Classes that process tag characters are also configured by default, so as long as there is no need to process custom tag characters, you can use the generated instance with no changes required. In addition, if you set no buffer to receive the fully uncompressed and formatted string, and then call a formatted `Printf()` function, 256 characters of memory is allocated from the stack. If you do not want to use the stack, or if a buffer with a capacity of more than 256 characters is needed, the application must allocate a

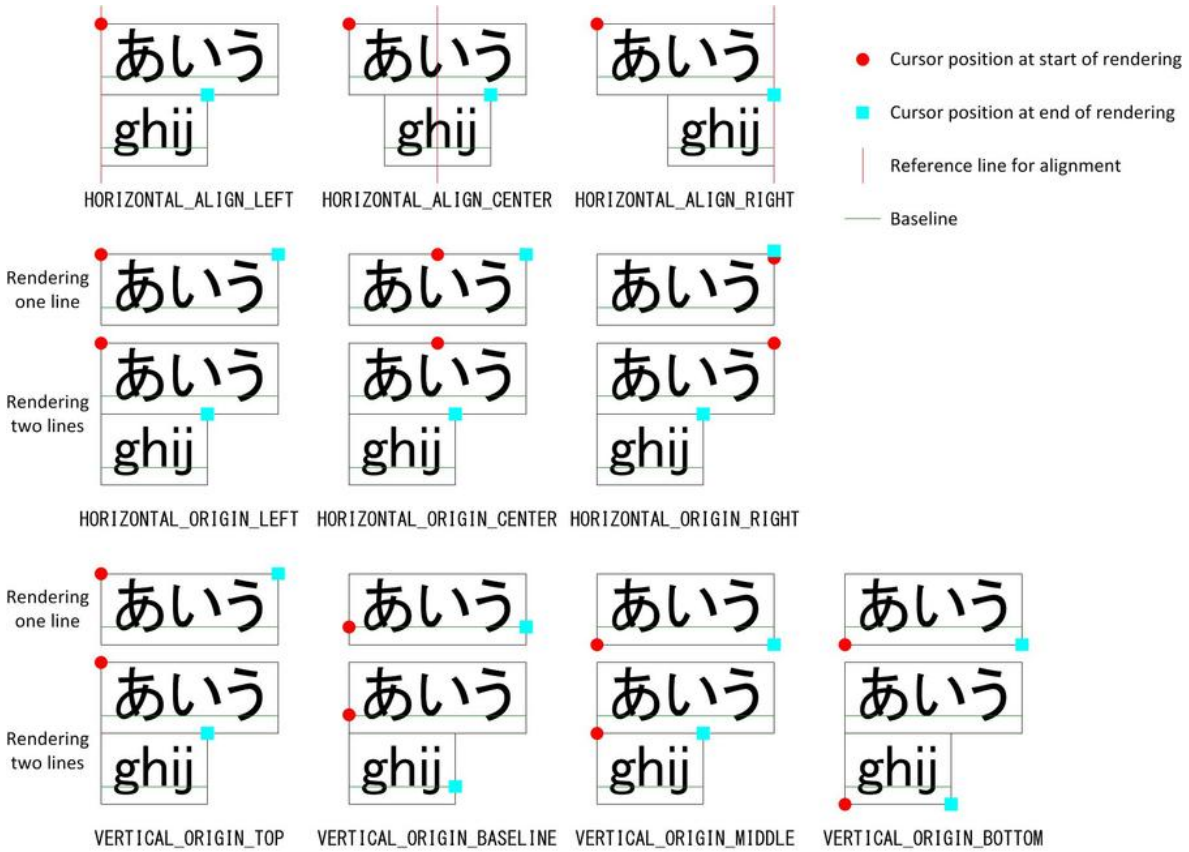
buffer and set it using the `SetBuffer()` function. The return value is a pointer to the previously set buffer.

The rendered strings are accumulated one character at a time as string display information. Use the `SetDispStringBuffer()` function to set the buffer for accumulating string information. (This buffer was allocated in 4.2.1.3 Allocating a Display String Buffer) for the `TextWriter` class. Use the `SetFont()` function to set the font resources used in rendering.

The `SetCursor` or `MoveCursor()` functions are used to set the display position (that is, the cursor position). `SetCursor` sets a new display position with no reference to a previous position. `MoveCursor` specifies the display position relative to the previous position. The x, y, and z coordinates can be set individually, or they can all be set at the same time. The default is (0.0, 0.0, 0.0).

The `SetDrawFlag()` function sets how the string is laid out in relation to the display position. This string layout is set by the bitwise OR of three flags: two flags (one each per vertical and horizontal axes) that specify where to place the origin of the rectangular region in which to render the string, and one flag (horizontal axis only) that specifies which direction to align or justify the string toward. The flags are defined by `nn::font::PositionFlag` enumerators. The default string layout setting is `HORIZONTAL_ALIGN_LEFT | HORIZONTAL_ORIGIN_LEFT | VERTICAL_ORIGIN_TOP`. Figure 4-3. illustrates how string layout changes as individual flags are changed from the default settings.

**Figure 4-3. Changes in String Layout With Different Flags**




Characters are scaled up or scaled down with the `SetScale()` or `SetFontSize()` functions. `SetScale` sets the width and height of characters as a multiple of the font's width and height values. `SetFontSize` sets pixel values for the width and height of the characters to display.

Scaling also affects the ascent and descent. For this reason, the `GetFontAscent` and `GetFontDescent()` functions might return different values than those of the font. The default scaling is 100% (no change).

The `SetTextColor()` function sets the character color. In addition, the `SetGradationMode()` function can apply color gradation effects in the horizontal or vertical directions. The `SetColorMapping()` function can set a linear transform for the gradation effect, and the gradation's change in color can also be reversed depending on its settings. Note that the `SetAlpha()` function can set an additional alpha value separate from the character color. The default settings are: a character color of (255,255,255,255), no gradation effect, no linear transform, and an additional alpha value of 255.

The `SetLineHeight()` function sets the height of one line. This setting automatically adjusts the line spacing, not the linefeed width. Use the `SetLineSpace()` function to directly set the line spacing. Additionally, the `SetTabWidth()` and `SetCharSpace()` functions set the tab width and character spacing, respectively. The default settings are: `0` for the line spacing and character spacing, and a tab width of 4 characters wide.

To forcibly render strings at a fixed width and ignore the font's settings, use the `SetFixedWidth()` function to set the rendering width, and use `EnableFixedWidth(true)` to enable monospaced rendering. Specify `false` in the `EnableFixedWidth()` function to again disable monospaced rendering. The default setting is `false` (disabled).

The settings discussed until now are related to the display of text, and are set for the `TextWriter` class. Graphics-related settings for rendering text are set for the `RectDrawer` class.

The `DrawBegin()` function performs the initialization required to begin rendering. This function generates rendering commands that initialize the render mode, textures, and other items. This function also initializes internal variables, so if you plan to set the parallax with the `SetParallax()` function, you must do so after this function is called.

Set the projection matrix and view matrix with the `SetProjectionMtx()` function and `SetViewMtxForText()` function, respectively. The Font library assumes that the projection matrix has its origin at the upper left, with the dimensions of the screen in screen space identical to those of the LCD screen, and the positive directions of the y-axis and z-axis are reversed from their sign conventions in the CTR coordinate system. The view matrix is assumed to be a unit matrix.

Strings are rendered by the `TextWriter` class. When characters are rendered, they are temporarily accumulated in the display string buffer as string display information, and the `RectDrawer` class generates rendering commands from the accumulated information.

Use the `TextWriter::StartPrint()` function to declare the start of string rendering. When this is done, the display string buffer is cleared.

Render strings using either the `TextWriter::Print` or `TextWriter::Printf()` function. The former does not specify a format string, while the latter does specify a format string.

When string rendering is complete, use the `TextWriter::EndPrint()` function to declare the end of string rendering. Thereafter, the `RectDrawer::BuildTextCommand()` function generates rendering commands based on the accumulated information and the `TextWriter::UseCommandBuffer()` function sends the rendering commands to the command

buffer.

Finally, call the `RectDrawer::DrawEnd()` function to kick off the rendering commands and actually render the characters in the render buffer.

If there is no change to the content or display position of the string to render, you can get the same rendering results without any need to execute again the sequence of functions from the `TextWriter::StartPrint()` function to the `TextWriter::EndPrint()` and `RectDrawer::BuildTextCommand()` functions. You can also render a string again and change only the character color, as long as the string was previously rendered in a solid color. However, this is possible only if the accumulated string display information has been saved.

The following code is the relevant portion of the sample demo.

**Code 4-7. Matrix Settings and Drawing Strings**

```
//------------------------------------------------------------------------
//! @brief      Sets the modelview matrix and projection matrix for the display
string.
//!
//! @param[in]  pDrawer         Pointer to the RectDrawer object.
//! @param[in]  width             Screen width.
//! @param[in]  height            Screen height.
//------------------------------------------------------------------------
void
SetupTextCamera(
    nn::font::RectDrawer*   pDrawer,
    int                     width,
    int                     height
)
{
    // Set the projection matrix to orthogonal projection.
    {
        // Set as the upper-left origin, with the y-axis and z-axis in the
reverse orientation.
        nn::math::MTX44 proj;
        f32 znear   = 0.0f;
        f32 zfar    = -1.0f;
        f32 t       = 0;
        f32 b       = static_cast<f32>(width);
        f32 l       = 0;
        f32 r       = static_cast<f32>(height);
        nn::math::MTX44OrthoPivot(
            &proj, l, r, b, t, znear, zfar, nn::math::PIVOT_UPSIDE_TO_TOP);
        pDrawer->SetProjectionMtx(proj);
    }

    // Set the modelview matrix to an identity matrix.
    {
        nn::math::MTX34 mv;
        nn::math::MTX34Identity(&mv);
        pDrawer->SetViewMtxForText(mv);
    }
}

//------------------------------------------------------------------------
//! @brief Draw ASCII characters.
//!
//! @param[in]  pDrawer         Pointer to the RectDrawer  object.
//! @param[in]  pDrawStringBuf  Pointer to the DispStringBuffer object.
//! @param[in]  pFont             Pointer to the font.
//! @param[in]  width             Screen width.
//! @param[in]  height            Screen height.
//------------------------------------------------------------------------
void
```

```
DrawAscii(
    nn::font::RectDrawer*       pDrawer,
    nn::font::DispStringBuffer* pDrawStringBuf,
    nn::font::ResFont*          pFont,
    int                         width,
    int                         height
)
{
    nn::font::TextWriter writer;
    writer.SetDispStringBuffer(pDrawStringBuf);
    writer.SetFont(pFont);
    writer.SetCursor(0, 0);

    // The strings are not changed, so the command for drawing strings is only
created once.
    if (! s_InitAsciiString)
    {
        writer.StartPrint();
            (void)writer.Print("DEMO:  ResFont\n");
            (void)writer.Print("\n");
            // Display ASCII character samples
            (void)writer.Print("All ASCII Character listing:\n");
            (void)writer.Print("\n");
            (void)writer.Print(" !\"#$%&'()*+,-./\n");
            (void)writer.Print("0123456789:;<=>?\n");
            (void)writer.Print("@ABCDEFGHIJKLMNO\n");
            (void)writer.Print("PQRSTUVWXYZ[\\\]^_\n");
            (void)writer.Print("`abcdefghijklmno\n");
            (void)writer.Print("pqrstuvwxyz{|}~\n");
        writer.EndPrint();
        pDrawer->BuildTextCommand(&writer);

        s_InitAsciiString = true;
    }

    // Character color can be changed without regenerating the string rendering
commands.
    writer.SetTextColor(nn::util::Color8(s_Color, 255, s_Color, 255));
    s_Color++;

    pDrawer->DrawBegin();
        SetupTextCamera(pDrawer, width, height);
        writer.UseCommandBuffer();
    pDrawer->DrawEnd();
}
```

## 4.2.1.6. Changing the GPU Settings With the Font Library

When fonts are rendered using the `RectDrawer` class, various GPU settings are changed including the texture sampler type and the vertex attribute load addresses. For this reason, after rendering fonts, have the application validate all of the GPU states and reset all of the GPU settings, although in some cases this processing might be redundant.

This section describes the GPU settings configured when fonts are rendered, as reference material for reducing the cost of resetting the GPU.

### Combiner Settings

Combiners 3 through 5 are used and set as follows. Some settings are different depending on

whether the sheet's texture format has an alpha component. Note also that the character color is expressed using the vertex color settings.

**Table 4-4. Combiner 3 Settings**

| Setting | Color | Alpha |
|---|---|---|
| Source 0 | GL_TEXTURE0 | GL_TEXTURE0 |
| Source 1 | GL_CONSTANT | GL_CONSTANT |
| Source 2 | GL_CONSTANT | GL_CONSTANT |
| Operand 0 | (No alpha) GL_SRC_COLOR<br>(With alpha) GL_ONE_MINUS_SRC_COLOR | GL_SRC_ALPHA |
| Operand 1 | GL_SRC_COLOR | GL_SRC_ALPHA |
| Operand 2 | GL_SRC_COLOR | GL_SRC_ALPHA |
| Combine | GL_MODULATE | GL_MODULATE |
| Scale | 1.0 | 1.0 |
| Constant color | White (1.0, 1.0, 1.0, 1.0) | |

**Table 4-5. Combiner 4 Settings**

| Setting | Color | Alpha |
|---|---|---|
| Source 0 | GL_TEXTURE0 | GL_TEXTURE0 |
| Source 1 | GL_CONSTANT | GL_CONSTANT |
| Source 2 | GL_PREVIOUS | GL_PREVIOUS |
| Operand 0 | (No alpha) GL_ONE_MINUS_SRC_COLOR<br>(With alpha) GL_SRC_COLOR | GL_ONE_MINUS_SRC_ALPHA |
| Operand 1 | GL_SRC_COLOR | GL_SRC_ALPHA |
| Operand 2 | GL_SRC_COLOR | GL_SRC_ALPHA |
| Combine | GL_MULT_ADD_DMP | GL_MULT_ADD_DMP |
| Scale | 1.0 | 1.0 |
| Constant color | Black (0.0, 0.0, 0.0, 0.0) | |

**Table 4-6. Combiner 5 Settings**

| Setting | Color | Alpha |
|---|---|---|
| Source 0 | GL_PRIMARY_COLOR | GL_PRIMARY_COLOR |
| Source 1 | GL_PREVIOUS | GL_PREVIOUS |
| Source 2 | GL_PREVIOUS | GL_PREVIOUS |
| Operand 0 | GL_SRC_COLOR | GL_SRC_ALPHA |
| Operand 1 | GL_SRC_COLOR | GL_SRC_ALPHA |
| Operand 2 | GL_SRC_COLOR | GL_SRC_ALPHA |
| Combine | GL_MODULATE | GL_MODULATE |

| | | |
|---|---|---|
| Scale | 1.0 | 1.0 |
| Constant color | Not yet set. Undefined. | |

#### Reserved Fragment Shader Settings

The settings of the following reserved fragment shader features are changed.

**Table 4-7. Reserved Fragment Shader Settings**

| Feature | Setting |
|---|---|
| Per-fragment operations mode | Standard mode (`GL_FRAGOP_MODE_GL_DMP`) |
| Fragment lighting | Disabled |
| Shadows (shadow textures) | Disabled |
| Fog ( and gas) | Disabled |
| Alpha test | Disabled |
| Blending | Enabled |

#### Texture Settings

Only texture 0 is used, and it is set to `GL_TEXTURE_2D`. The texture image address and resolution, and filter settings and other settings are changed.

#### Shader Settings

Data required for rendering is set in the integer register i0 and the floating-point constant registers c0 through c95.

The shader program outputs the vertex coordinates, vertex color, texture coordinate 0, texture coordinate 1, and texture coordinate 2, but geometry shaders cannot be used.

## 4.2.2. Customizing the Processing of Tag Characters

By customizing classes that inherit the `nn::font::TagProcessorBase` class, it is possible to implement custom processing of tag characters (`0x0000` to `0x001F`).

When a tag character appears as a character to render in the middle of rendering a string, the `TextWriter` class passes the character to the `TagProcessor` class set in the `TextWriter` for processing. By default, the `TextWriter` class is configured to have a `TagProcessor` class that processes the tab character (`0x0009`) and the newline character (`0x000A`), but you can use the `SetTagProcessor()` function to set a customized `TagProcessor` class.

When you use customized tag character processing, override the `Process()` and `CalcRect()`

functions.

**Code 4-8. Functions to Override When Using Customized Tag Character Processing**

```
virtual Operation Process( u16                         code,
                           PrintContext<CharType>*     pContext);
virtual Operation CalcRect(util::Rect*                 pRect,
                           u16                         code,
                           PrintContext<CharType>*     pContext);
```

The `TextWriter` class adjusts the coordinates at which to render characters based on the values returned from these functions. Implement your application so that the `Process()` and `CalcRect()` functions both return the same value when they process the same tag character.

**Table 4-8. Definitions of Return Values of Tag Character Processing**

| Definition | Processing Performed by TextWriter |
|---|---|
| `OPERATION_DEFAULT` | Inserts no space before the next character if it is the first character in a line, but otherwise always inserts space before the next character. |
| `OPERATION_NO_CHAR_SPACE` | Never inserts a space before the next character. |
| `OPERATION_CHAR_SPACE` | Always inserts a space before the next character. |
| `OPERATION_NEXT_LINE` | Performs newline processing. Only adjusts the x-coordinate; the y-coordinate must be adjusted when processing tags. |
| `OPERATION_END_DRAW` | Ends string rendering in the middle of a string. |

A pointer to the `TextWriter` class is included in the members of the `pContext` argument, so it is possible to use tag characters for special processing, such as changing the character color.



# 5. Creating Fonts

Font resources used by the Font library can be created using `ctr_FontConverter` (or the command-line version `ctr_FontConverterConsole`) from fonts installed on a PC or from image data.

**Warning:**CTR-SDK does not include a license to any font.

The `ctr_FontConverter` program uses fonts installed on a PC or image data to create font resources for the display of text using the font library. However, a license to use these fonts is needed to sell software that uses these fonts. You must always obtain all the appropriate licenses for each of your software titles.

Not only can `ctr_FontConverter` create font resources, but it also can output fonts as image data. The possible image data formats are BMP format and TGA format. Image data in these formats can be used to create font resources. In other words, you can output image data, revise or tweak it, and recreate the font resource from that data, and you can combine image data to create a font resource that contains multiple fonts.

# 5.1. Files Used to Create Font Resources

The following files are used when `ctr_FontConverter` creates font resources.

- Character filter files
- Glyph group files
- Character order files

These files are all written in XML (eXtensible Markup Language), and generally share a common structure.

## 5.1.1. Common Structures

Although the name of the root element is different in the three types of files, each file has a common structure that contains the version attribute, the head element, and the body element. Although the structure of the head element is common among the files, the structure of the body element differs for each of the files.

Table 5-1 below lists the elements in the common portions. The elements and attributes indicated in **bold** are required.

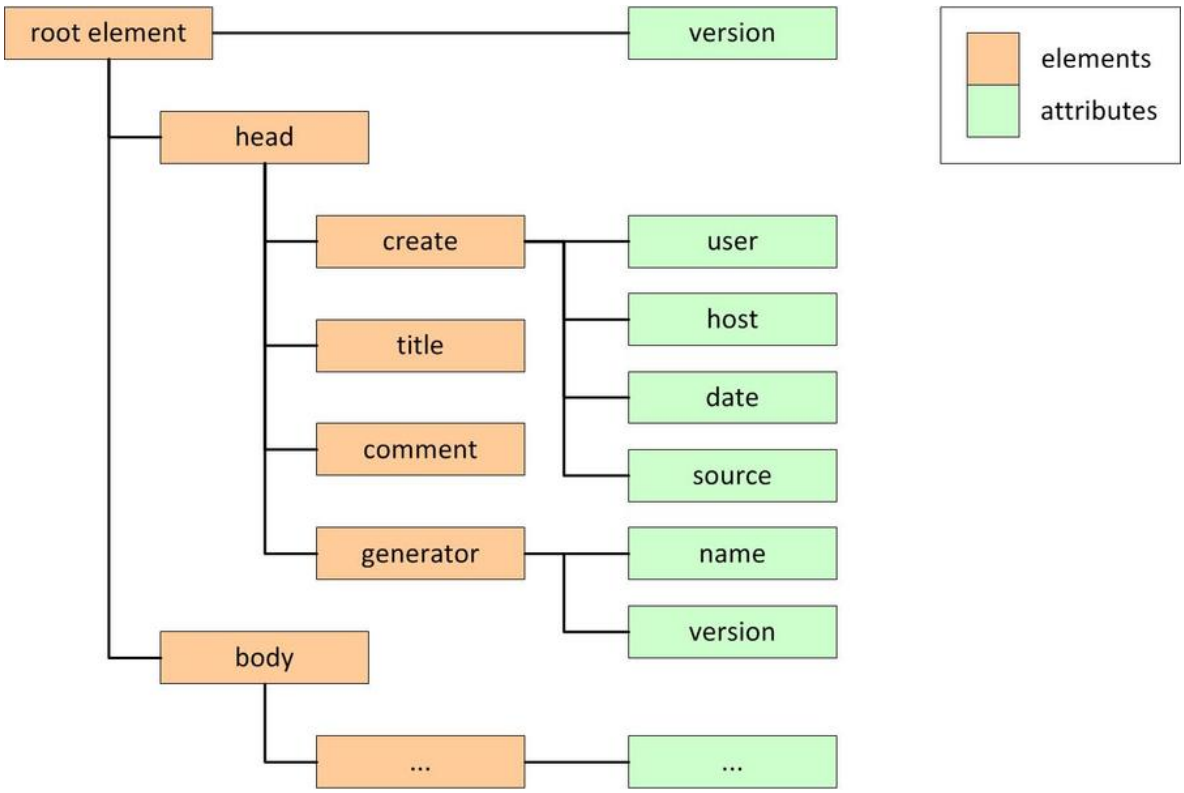
**Table 5-1. Elements in the Common Portions of the XML Files**

| Element | Elements It May Contain (Top Cell) Attributes (Bottom Cell) | Description |
|---|---|---|
| **Root element** | HEAD, BODY | For the root element, the name of the element is different in each file. The elements it may contain and its attributes are common to all files. |
| | **version** | |
| HEAD | CREATE, TITLE, COMMENT, GENERATOR | Contains the information describing the file itself (the header). This element has a common structure in all files. |

| | | |
|---|---|---|
| | None. | |
| CREATE | None. | Contains information set when the file was created.<br>The **user** attribute contains the user name on the PC where the file was created. The **host** attribute contains the name of the PC. The **date** attribute contains the file creation date and time in ISO 8601 extended format.<br>The **source** attribute contains the filename of the source data file if a source data file exists. |
| | user, host, date, source | |
| TITLE | None. | Contains the title of the file.<br>This title is sometimes displayed as a selectable item in the GUI version of `ctr_FontConverter`. |
| | None. | |
| COMMENT | None. | Contains comments from the creator of the file. |
| | None. | |
| GENERATOR | None. | Contains information about the application that created the file.<br>The **name** attribute contains a string that identifies the application.The **version** attribute contains the application version string. |
| | name, version | |
| **body** | Depends on the file | Contains information comprising the body of the file. |
| | None. | |

Figure 5-4 illustrates the hierarchy of elements.

**Figure 5-1. Hierarchy of Elements in the Common Portions of the XML Files**



# 5.1.2. Character Filter Files

The character filter file (extension `.xllt`) is used for the creation of a compact font resource that contains only the required characters. When the font resource is converted into a font, it is possible to suppress the output of all characters except those specified in this file. This file is unnecessary when all characters contained in the pre-conversion data are to be included in the font.

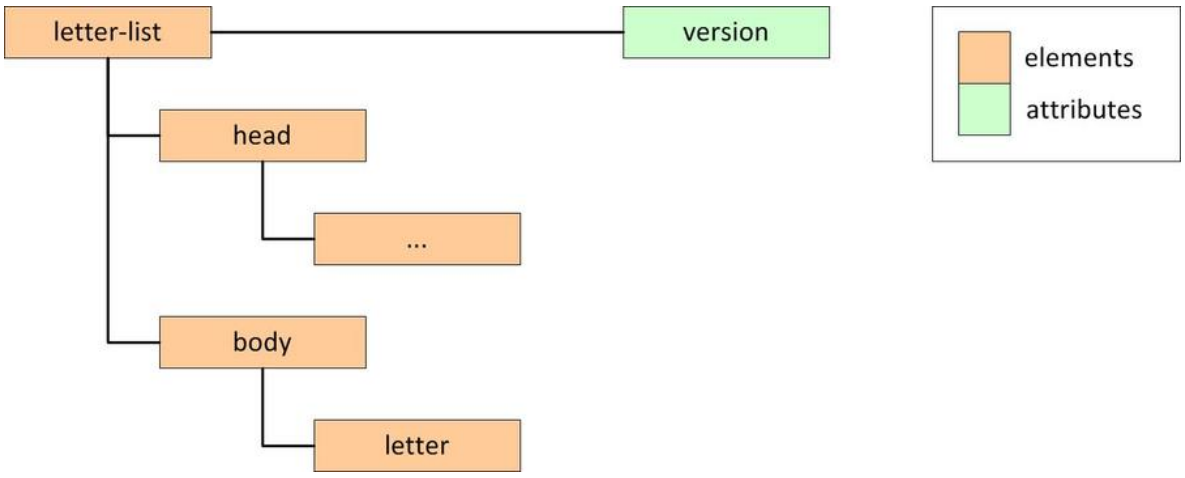
The root element is LETTER-LIST and `1.0` is stored in the **version** attribute. The BODY element contains only one element: letter.

**Table 5-2. Elements in the Character Filter File (Excluding the Common Portions)**

| Element | Elements It May Contain (Top Cell) Attributes (Bottom Cell) | Description |
|---|---|---|
| **LETTER-LIST** | HEAD, BODY | The root element.<br>The **version** attribute contains `1.0`. |
| | **version** | |
| **BODY** | **LETTER** | Contains information comprising the body of the file.<br>The only element it can contain is the single element LETTER. |
| | None. | |
| **LETTER** | None. | Defines the characters to output.<br>The characters to output are specified by directly coding those characters. However, white-space characters (half-width spaces and tab characters) are ignored, and the half-width space is always included in the conversion results. |
| | None. | |

Figure 5-4 illustrates the hierarchy of elements.

**Figure 5-2. Hierarchy of Elements in the Character Filter File**




The code below is the content of a sample character filter file (`sample.xllt`).

**Code 5-1. Sample Character Filter File (sample.xllt)**

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE letter-list SYSTEM "letter-list.dtd">
<letter-list version="1.0">
    <head>
        <create user="sample" date="2005-02-18T10:51:13" />
        <title>XLLT Sample</title>
        <comment>Sample Character Filter File</comment>
    </head>
    <body>
        <letter>
            FontConverter
            あいうえかきくけ
            任天堂
        </letter>
    </body>
</letter-list>
```

## 5.1.3. Glyph Group Files

The glyph group file (extension **.xggp**) is used for the creation of archived fonts (compressed fonts). By specifying a group set (or specifying a file using the command argument –op in the command-line version), you can create a font resource containing the group information set in this file.

The root element is GLYPH-GROUPS and **1.0** is stored in the version attribute. The BODY element contains only one element: letter.

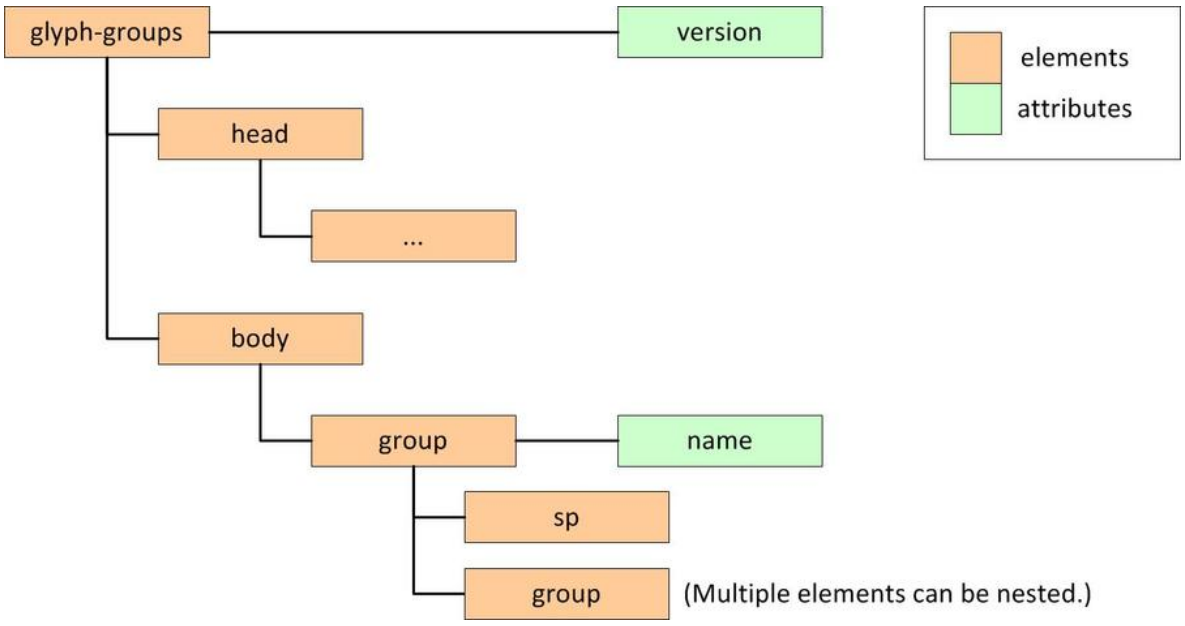
**Table 5-3. Elements in the Glyph Group File (Excluding the Common Portions)**

| Element | Elements It May Contain (Top Cell) Attributes (Bottom Cell) | Description |
|---|---|---|
| **GLYPH-GROUPS** | HEAD, BODY | The root element.<br>The **version** attribute contains **1.0**. |
| | **version** | |
| **BODY** | **GROUP** | Contains information comprising the body of the file.<br>The only element it may contain is the single element group. |
| | None. | |
| **GROUP** | SP, GROUP | Defines the glyph group.<br>The enumerated characters are contained in a glyph group having the name specified in the **name** attribute. Multiple group elements can be nested in a GROUP element.<br>The only characters that can be used in the **name** attribute are the half-width alphanumeric characters (0–9, a–z, A–Z) and the underscore character (_).<br>Within the same file, all GROUP elements must have unique **name** attributes. |
| | **name** | |
| | None. | Half-width spaces coded directly within GROUP elements are ignored, so this |

| | | |
|---|---|---|
| SP | None. | element is used to specify the half-width space within GROUP elements. |

Figure 5-4 illustrates the hierarchy of elements.

**Figure 5-3. Hierarchy of Elements in the Glyph Group File**




The code below is excerpted from a sample glyph group file (`sample.xggp`).

**Code 5-2. Sample Glyph Group File (sample.xggp)**

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE glyph-groups SYSTEM "glyph-groups.dtd">
<glyph-groups version="1.0">
    <head>
        <create user="sample" date="2006-09-05T14:50:23" />
        <title>sample</title>
    </head>
    <body>
        <group name="all">
            <group name="ascii">
                <sp/> ! " # $ % & ' ( ) * + , - . /
                (omitted)
                p q r s t u v w x y z { | } ~
            </group>
            <group name="european">
                € , ƒ „ ... † ‡
                ˆ ‰ Š ‹ Œ Ž
                (omitted)
            </group>
        (omitted)
        </group>
    </body>
</glyph-groups>
```

## 5.1.4. Character Order Files

The character order file (extension `.xlor`) is necessary when creating a font resource from image data or when outputting a font as image data. When used for input of image data, the order of the characters rendered in the image data must match the order of characters set in this file. When used for output of image data, the characters are rendered in the order set in this file, so this file also serves as a filter.

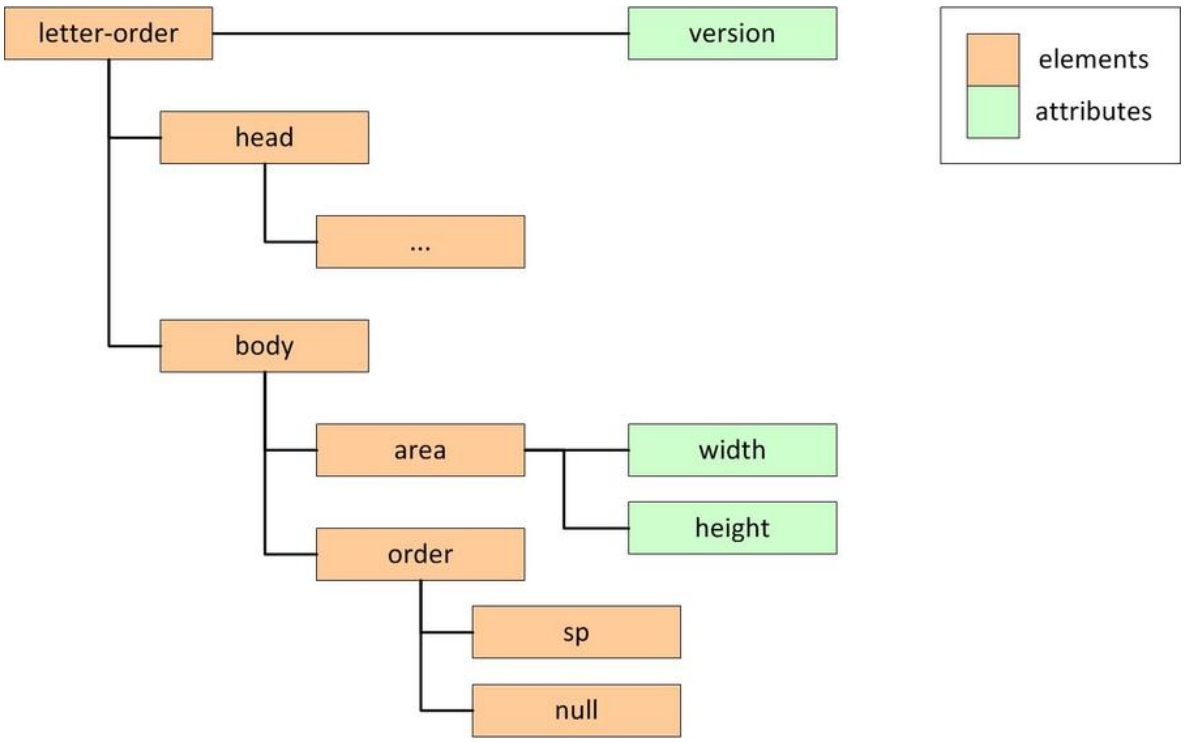
The root element is LETTER-ORDER and `1.0` or `1.1` is stored in the version attribute. The BODY element contains the AREA and ORDER elements.

**Table 5-4. Elements in the Character Order File (Excluding the Common Portions)**

| Element | Elements It May Contain (Top Cell) Attributes (Bottom Cell) | Description |
|---|---|---|
| **LETTER-ORDER** | HEAD, BODY | The root element.<br>The **version** attribute contains `1.0`.<br>If `1.0` is specified as the version attribute, no error results if multiple characters that have the same character code are coded within the ORDER element. However, if multiple glyphs that have the same character code are present, the first-coded glyph among them is used. Accordingly, edits to the other glyph images might not be reflected in the font resource. If `1.1` is specified as the **version** attribute, an error results if multiple characters that have the same character code are coded within the ORDER element, so we normally recommend that `1.1` be specified. |
| | **version** | |
| **body** | AREA, ORDER | Contains information comprising the body of the file.<br>It can contain the AREA element and the ORDER element. |
| | None. | |
| AREA | None. | Defines the font image contained in the image data using the number of characters in the horizontal and vertical directions.<br>The **width** attribute defines the number of characters in the horizontal direction. If this attribute is omitted, the default is 16.<br>The **height** attribute defines the number of characters in the vertical direction. If this attribute is omitted, the default is a value sufficient to output all the characters specified in the ORDER element. |
| | width, height | |
| **ORDER** | SP, NULL | Defines the characters to output and their order.<br>The characters enumerated in this element are output to the font resource in the order defined within it. |
| | None. | |
| SP | None. | Half-width spaces coded directly within the ORDER element are ignored, so this element is used to specify half-width spaces in the ORDER element. |
| | None. | |
| NULL | None. | This element causes the output position to skip forward by one character. Use this element if you do not want to output the block in that position to the font resource. |
| | None. | |

Figure 5-4 illustrates the hierarchy of elements.

**Figure 5-4. Hierarchy of Elements in the Character Order File**




The code below is excerpted from a sample character order file (`cp1252.xlor`).

**Code 5-3. Sample Character Order File (cp1252.xlor)**

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE letter-order SYSTEM "letter-order.dtd">
<letter-order version="1.1">
    <head>
        <create user="sample" date="2005-02-18T10:51:13" />
        <title>European Language (CodePage 1252 / Latin-1)</title>
        <comment>Windows Code Page 1252 character set. It is super set of ISO
8859-1 (Latin-1).</comment>
    </head>
    <body>
        <area width="16" />
        <order>
            <sp/> ! " # $ % & ' ( ) * + , - . /
            0 1 2 3 4 5 6 7 8 9 : ; < = > ?
            (omitted)
            € <null/> ‚ ƒ „ ... †
            ‡ ˆ ‰ Š ‹ Œ <null/>
            Ž <null/>
            (omitted)
        </order>
    </body>
</letter-order>
```

## 5.1.4.1. Notes on Coding

`ctr_FontConverter` uses Unicode as the character encoding for its internal processing. Accordingly, it uses Unicode to determine whether character codes are identical. For example, in

CP1252 and Shift-JIS, the ASCII character codes match, and some extended Latin characters and half-width Japanese katakana characters are assigned to the same character code in these two different encodings. In contrast, in Unicode different character codes are allocated to the extended Latin characters and half-width Japanese katakana characters, so `ctr_FontConverter` determines them to not be the same character. Yet for the 16 characters listed in Table 5-5, although different character codes are allocated to these characters in CP1252 and Shift-JIS, the same character code is allocated in Unicode, so `ctr_FontConverter` unfortunately determines them to be identical. To avoid this problem, in the character order files `cp1252_JIS_X0201_X0208_012.xlor` and `cp1252_JIS_X0201_X0208_012_94.xlor` provided with the SDK, these 16 characters on the Shift-JIS side are replaced with `<null/>`.

**Table 5-5. Characters Mapped to the Same Character Code in Unicode**

| Glyph | Character Code in CP1252 | Character Code in Shift-JIS | Character Code in Unicode |
|---|---|---|---|
| ´ | 0xB4 | 0x814C | U+00B4 |
| ¨ | 0xA8 | 0x814E | U+00A8 |
| … | 0x85 | 0x8163 | U+2026 |
| ' | 0x91 | 0x8165 | U+2018 |
| ' | 0x92 | 0x8166 | U+2019 |
| " | 0x93 | 0x8167 | U+201C |
| " | 0x94 | 0x8168 | U+201D |
| ± | 0xB1 | 0x817D | U+00B1 |
| × | 0xD7 | 0x817E | U+00D7 |
| ÷ | 0xF7 | 0x8180 | U+00F7 |
| ° | 0xB0 | 0x818B | U+00B0 |
| § | 0xA7 | 0x8198 | U+00A7 |
| ‰ | 0x89 | 0x81F1 | U+2030 |
| † | 0x86 | 0x81F5 | U+2020 |
| ‡ | 0x87 | 0x81F6 | U+2021 |
| ¶ | 0xB6 | 0x81F7 | U+00B6 |

## 5.2. Creating Font Resources From Fonts Installed on a PC

`ctr_FontConverter` can create font resources from fonts installed on a PC. To exclude the possibility of using unlicensed fonts in products, the font resources are created using automatic font linking disabled, so that characters not included in the specified font are not included in the font resource.

When text is viewed in an ordinary text editor, automatic font linking enables characters not included in the font to sometimes be supplemented by characters from another font. The Windows application

Character Map can be used to confirm exactly what characters are actually contained in a font.

## 5.3. Creating Font Resources From Image Data

`ctr_FontConverter` can create font resources from the combination of character order files and image data on which the font images have been rendered. The image data that can be used as input to `ctr_FontConverter` must be in the following format.

**Table 5-6. Formats of Image Data Supported by ctr_FontConverter**

| Format | Scope of Support |
|---|---|
| Indexed color BMP | Per pixel: 1, 4, or 8 bits (2, 16, or 256 colors) |
| Direct color BMP | Per pixel: 24 bits (RGB8) or 32 bits (RGBA8). |
| ColorMap type TGA | Per pixel: index is 8 or 16 bits (256 or 65,536 colors), and palette is 24 bits (RGB8) or 32 bits (RGBA8). Run-length encoding compressed data is also supported. |
| TrueColor type TGA | Per pixel: 24 bits (RGB8) or 32 bits (RGBA8). Run-length encoding compressed data is also supported. |

`ctr_FontConverter` performs the following conversions depending on the color format and image data format specified as image data input options. Note that in intensity-format fonts, `ctr_FontConverter` inverts the luminance values of the image data. In other words, the closer a pixel is to black the higher its luminance value becomes, and the closer a pixel is to white the lower its luminance value becomes.

**Table 5-7. Color Formats and Processing Performed in Conversion**

| Color | Conversion Processes |
|---|---|
| A4 | In the case of indexed color, the index value is converted to 4 bits.<br>In the case of direct color, the average of the RGB components is converted to 16-level grayscale. |
| A8 | In the case of indexed color, the index value is converted to 8 bits.<br>In the case of direct color, the average of the RGB components is converted to 256-level grayscale. |
| LA4 | In the case of indexed color, the index value converted to 4 bits is used as the luminance value.<br>In the case of direct color, the average of the RGB components converted to 16-level grayscale is used as the luminance value.<br>Regardless of the color format of the image data, the per-pixel alpha component converted to 4 bits is used as the alpha value. |
| | In the case of indexed color, the index value converted to 8 bits is used as the luminance value.<br>In the case of direct color, the average of the RGB components converted to 256-level |

| | |
|---|---|
| LA8 | grayscale is used as the luminance value.<br>Regardless of the color format of the image data, the per-pixel alpha component converted to 8 bits is used as the alpha value. |
| RGB565 | The RGB components are converted to 5, 6, and 5 bits, respectively, and the alpha component is not used. |
| RGB5A1 | The RGB components are converted to 5 bits each, and only the most significant bit of the alpha component is used. |
| RGBA4 | The RGBA components are converted to 4 bits each. |
| RGB8 | The RGB components are converted to 8 bits each, and the alpha component is not used. |
| RGBA8 | The RGBA components are converted to 8 bits each. |

#### Linear Interpolation Correction

When the Font library renders characters, glyphs are displayed as textures applied to polygons. This enables hardware-based texture linear interpolation to be used to scale up characters. However, if the font format has an alpha channel and the pixel's alpha value is `0`, texture linear interpolation looks up the pixel color instead, which sometimes causes unintended display results.

With `ctr_FontConverter`, you can apply linear interpolation correction when you convert image data into a font resource. In linear interpolation correction, the color of pixels that are completely transparent (have an alpha value of `0`) is overwritten with the average pixel color of all pixels that are not completely transparent (have a nonzero alpha value) among the eight adjacent pixels. The alpha value is not overwritten. Correcting glyphs in this manner resolves the unintended display problems caused by linear interpolation.

## 5.3.1. Blocks

A *block* is a subsection of the image data that contains a cell in which a glyph image has been rendered and a *width-line region* that indicates the character width. Each character coded in the character order file corresponds to one block, and the image data used for input to `ctr_FontConverter` must have these blocks packed one after another with no gaps in between.

The following limitations apply to the image data used for input.

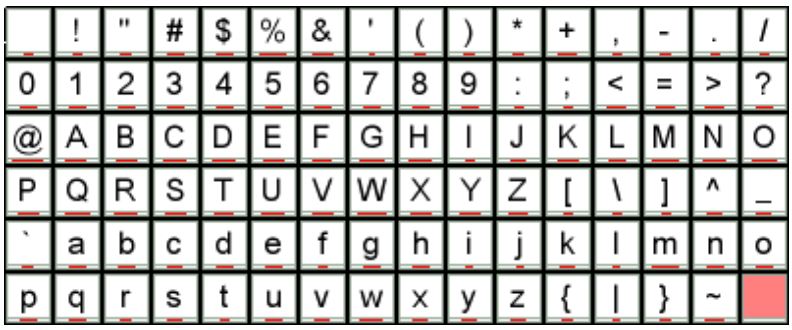
> width of one block × number of blocks in the horizontal direction = width of the image
>
> height of one block × number of blocks in the vertical direction = height of the image

In other words, the blocks in the image are aligned with no gaps in between, and there must be no empty space or spacing between blocks.

In fact, the number of blocks in the horizontal (or vertical) directions are defined in the character order file. The tool compares these numbers to the dimensions of the image as determined from the image data, and calculates the dimensions of the individual blocks. These calculated block dimensions must be integers or the image data cannot be used for input.

The figure below is an example of image data.

**Figure 5-5. Example of Image Data (Number of Blocks: 16×6)**




Each block contains a cell, which is the region in which the glyph image has been rendered. Below the cell, there is a one-pixel blank area, and below that there is a *width-line region* having a height of one pixel. The width-line in this region defines the character width and the relative position of the glyph image within the cell. There is a one-pixel-wide blank frame surrounding the cell and the width-line region, and a one-pixel-wide grid frame surrounding the blank frame. Accordingly, the block boundary is the boundary of that grid frame against the grid frame of the adjacent block. The grid frames are not checked when image data is converted, but the blank frames must contain a solid color or the tool will determine that a glyph image has overlapped the edge of the cell.

The height of the width-line region and the number of pixels in the blank frame and grid frame are fixed, so the size of a cell is always determined as follows.

cell width = block width – 4 pixels

cell height = block height – 6 pixels

Figure 5-6 below is a schematic drawing of a block.

**Figure 5-6. Schematic Drawing of a Block**

## 5.3.2. Cells and Width-Line Regions

Only the glyph image is placed within the cell. `ctr_FontConverter` detects the smallest rectangle present within the cell that contains all pixels with nonzero alpha values and all pixels whose color is not white (255, 255, 255), and handles it as the glyph image. For this reason, even if the cell itself is large, the white transparent spaces surrounding the glyph image are ignored and have no effect on the created font. If the image data does not include an alpha channel, the smallest rectangle containing non-white pixels is handled as the glyph image.

Only a single line segment called the *width-line* is drawn in the width-line region. The width-line defines the character width and the width of the spaces to the left and right of the character. The length of the width-line is taken as exactly equivalent to the character width, and any abnormality in this line (such as being discontinuous and broken into more than one segment) causes an error. The character width can be made smaller than the width of the glyph image, and in this case the character is displayed on 3DS overlapping with the previous character. You can move the glyph image left or right within the cell without affecting the output font as long as the width-line is also moved and maintains the same positional relationship with the glyph image.

If the size of the glyph image is `0` and the length of the width-line is also `0`, or if the interior of the cell is a solid single color and the length of the width-line is `0`, that glyph is not output. You can use this fact to control what glyphs are output instead of using a character filter file. Conversely, if a glyph is not passed for output despite being specified for output in the character filter file, the character may have been dropped unintentionally, and a warning is displayed.

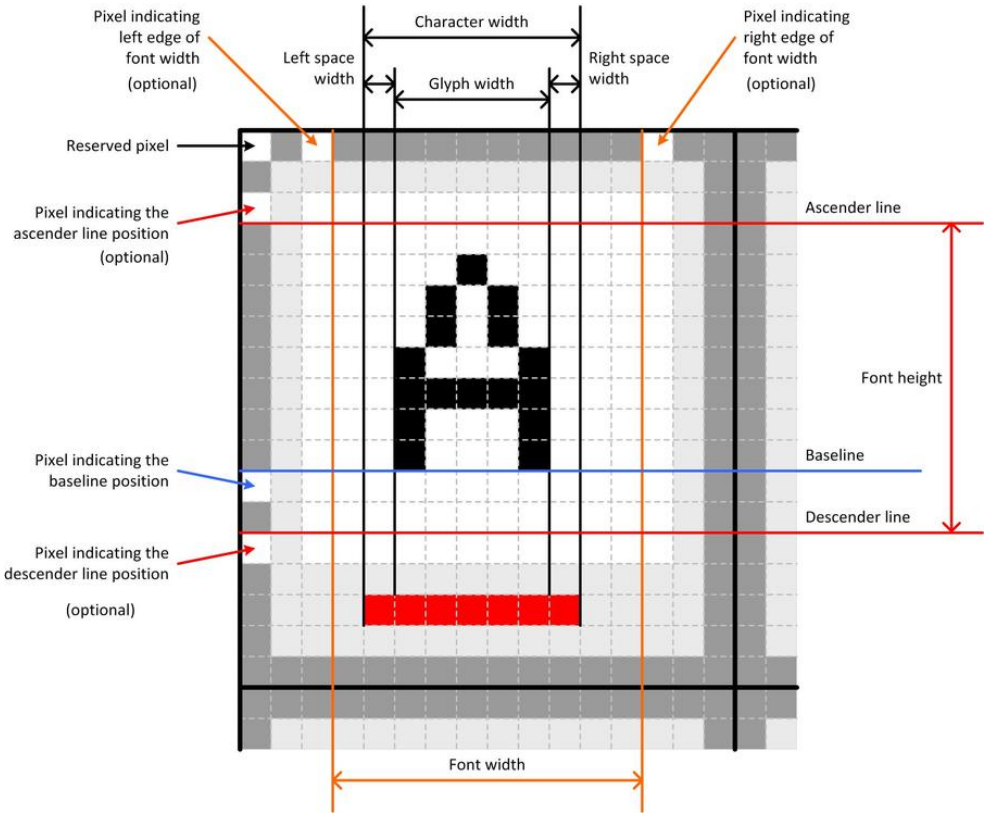
## 5.3.2.1. Layout Information

The block at the upper-left corner of the image data contains several single-pixel points that specify parameters for string layout when strings are rendered. These single-pixel dots are the layout information. The four values serving as the parameters for layout are the baseline, the ascender line, the descender line, and the font width. These four values are specified with five points. These values are common to the entire font and cannot be specified for individual characters.

All of the dots are located in the grid frame and are white (255, 255, 255) if a grid has been rendered, or are rendered in the grid color if no grid has been rendered. Although the pixel that indicates the baseline position must always be present, the remaining four pixels may be omitted.

The single pixel at the extreme upper-left corner of the image is reserved for future use and must always be white (255, 255, 255). In addition, this single pixel can also be used as a color key for transparent color. If an alpha value is entered for this pixel, the transparent color is determined to be the color containing the alpha value.

**Figure 5-7. Block Layout Information**




### Baseline

The point indicating the baseline position is rendered in the left edge of the grid frame. The upper edge of this pixel indicates the baseline position. In other words, the baseline lies on the boundary between pixels. The baseline serves as the vertical reference position when different fonts are used together or when characters are scaled.

### Ascender Line and Descender Line

The points indicating the ascender line and descender line positions are also rendered on the left edge of the grid frame. Like the baseline, the ascender line and descender line lie on the boundary between pixels, with the lower edge of the ascender line's pixel indicating the position of the ascender line, and the upper edge of the descender line's pixel indicating the position of the descender line. The ascender line is treated as the upper edge of the string, while the descender line is treated as the lower edge of the string. The distance between the ascender line and descender line is called the font height, which is the reference size when scaling characters in the vertical direction.

The ascender line must always be positioned above the baseline, and the descender line must always be positioned below the baseline. However, it is possible for the baseline and the descender line to be specified by the same pixel, and in this case the descent is `0`.

The two points indicating the ascender line position and the descender line position are optional and may be omitted, with the remaining points being interpreted as illustrated in Table 5-8. If no ascender line position or descender line position is specified, the upper edge of the cell is treated as the ascender line and the lower edge of the cell is treated as the descender line.

When `ctr_FontConverter` is used to output image data, the pixels indicating the ascender line position and the descender line position are sometimes rendered and sometimes not rendered. They are never rendered in conversions where Windows fonts are used as input. If image data is used as input, they are rendered if the input image contains pixels indicating the ascender line position and the descender line position, but are not rendered if not. When conditions are right to render the pixels indicating the ascender line position and descender line position, but the cell is too small to represent these pixels, the cell is automatically enlarged.

**Table 5-8. Number of Points and Their Interpretation**

| Number of Points | First Point from Top | Second Point from Top | Third Point from Top |
|---|---|---|---|
| 0 | Error | | |
| 1 | Baseline | – | |
| 2 | Ascender line | Baseline and descender line | – |
| 3 | Ascender line | Baseline | Descender line |
| 4 or more | Error | | |

### Font Width

The two points indicating the font width are rendered in the top edge of the grid frame. The two points represent the left edge and right edge of the width, respectively, so the number of pixels between these two pixels is the font width. These two points can be moved left or right with no effect on output as long as there is no change in the number of pixels between them. If the points indicating the font width are omitted, the width of the cell is taken as the font width. If used, both points must always be present (that is, you cannot omit just one of these points). The font width serves as the reference size for the tab width and for scaling characters in the horizontal

direction.

When `ctr_FontConverter` is used to output image data, the pixels indicating the font width are sometimes rendered and sometimes not rendered. They are never rendered in conversions where Windows fonts are used as input. They are always rendered in conversions where font resources are used as input. If image data is used as input, they are rendered if the input image contains pixels indicating the font width, but are not rendered if not. When conditions are right to render the pixels indicating the font width but the cell is too small to represent these pixels, the cell is automatically enlarged.

## 5.4. Creating Font Resources From Existing Font Resources

Font resources can also be created and image data can also be output from BCFNT or BCFNA files. This capability can be used for filtering or investigating the characters contained in the font resources, or for checking or editing their glyph images.

> **Warning:**Be sure to obtain licenses to the fonts contained in the source font resources.



# Revision History

## Version 1.1 (2015-11-05)

### Changes

- 3.2. ResFont Class
  - Added a definition name for the alignment restriction.

## Version 1.0 (2014-09-04)

## Additions and Changes

- Initial version.

# 3DS Programming Manual: Creating Applications

Version 1.3

**Nintendo Confidential**

This document contains confidential and proprietary information of Nintendo, and is protected under confidentiality agreements as well as the intellectual property laws of the United States and of other countries. No part of this document may be released, distributed, transmitted, or reproduced in any form , including by any electronic or mechanical means and by including within information storage and retrieval systems, without written permission from Nintendo.

---

©2015 Nintendo. All rights reserved.

All company and product names in this document are the trademarks or registered trademarks of their respective companies.

CTR-06-0237-002-D

# 1. Introduction

This document summarizes information related to creating 3DS applications. It is written for application developers.

## 1.1. Reading This Document

This document is intended to introduce information related to the creation of applications. For more information, see the related documentation as appropriate.

This document is structured as follows.

2. Types of Applications lists the types of 3DS applications that can be created and describes the characteristics of each.

3. Implementation Tips, 4. Preparing to Build, and 5. Building describe the basic procedures for creating an application.

6. Debugging explains how to debug developer applications.

7. Using DLLs (Dynamic Modules) describes the steps that must be followed when using dynamic modules in developer applications.

8. Developing Card Applications That Can Be Sold via Download describes the steps that must be followed when distributing the same application both as a card version and downloadable version.

## 1.2. Glossary

This section describes the terms used in this document. These terms may be used in ways that differ from their common meanings.

**Table 1-1. Glossary**

| Term | Description |
|---|---|
| Card-based application | A 3DS application stored on and run from a Game Card. Data is read from the Game Card when the application is executed. |
| Downloadable application | A 3DS application stored on and run from a Game Card. Data is read from the SD card when the application is executed. |
| CXI (CTR executable image) | CTR application executable image format. Also any file stored in this format, and the filename extension. |
| E-manual | An application instruction manual that is displayed on the 3DS system's screens. E-manuals are created by CTR-ManualTools. |
| CFA (CTR file archive) | A binary format handled as a data container on CTR systems. Also any file stored in this format, and the filename extension. |
| CCI (CTR card image) | The format of the final ROM image for card-based applications. Also any file stored in this format, and the filename extension. This includes a CXI within it. |
| CIA (CTR importable archive) | The format of the final ROM image for downloadable applications, the file that stores this image, and the extension of this file. Also any file stored in this format, and the filename extension. This includes a CXI within it. The client programs distributed through Download Play are also created with this format. |
| RSF (ROM specification file) | Specification file describing the parameters used when creating a CXI file. |

| | |
|---|---|
| Remaster version | Application version number described in the RSF file and then embedded in the CXI file. |
| Import | Storing a download application on an SD card and making it ready to run. |
| HOME Menu | Screen that displays when the 3DS is turned on. This screen is used to select applications to start. |
| Test menu | The development screen that appears when the PARTNER-CTR Debugger or CTR test unit is turned on. This screen has a minimal feature set and is used in place of the HOME Menu. |
| DevMenu | An application that can import CIA files to an SD card, display extended save data, and perform other operations used during development. |
| Archive | Individual file system operated on by the CTR-SDK's FS library. Corresponds to a drive on a Windows PC. |
| Save data region | Memory region allocated for storing save data. This region is allocated on an SD card for a downloadable application and can only be used on the system that created it. |
| Extended save data region | Data region on an SD card that is created and managed separately from save data. This can only be used on the system that created it. The extended save data created in this region is also called "extra data." |
| Module | A logical collection of executable code. |
| Dynamic module | A module that can be dynamically loaded into memory and executed. |
| Static module | The module that represents the core application and is executed first. Any application that does not use dynamic modules can be said to comprise only static modules. |
| DLL | The mechanism for implementing a dynamic module. It can also refer to a dynamic module itself. |
| CRS | Static module. Also any file stored in this format, and the filename extension. |
| CRO | Links dynamic modules. Also any file stored in this format, and the filename extension. |
| CRR | Loads dynamic modules. Also any file stored in this format, and the filename extension. This file can store one or more instances of CRO registration information. |
| Package version | The version of a dual-distribution application that is stored on a Nintendo 3DS Game Card and sold in retail stores. |
| Download version | The version of a dual-distribution application that is sold via download. |
| Revised version | After an application has been released, the new version application of the same title is released with some changes. |
| Patch | A mechanism to replace some of the elements of an application and the data used to make those replacements. |
| Standard application | An application that runs in standard mode when run on SNAKE. Such applications are Nintendo 3DS-specific software, including already released titles. |
| Enhanced application | An application that runs in enhanced mode when run on SNAKE. You can also run these applications on CTR. Running enhanced applications on CTR is equivalent to running on SNAKE in standard mode. |

# 2. Types of Applications

You can develop the following types of 3DS applications.

## 2.1. Card-Based Software

These applications combine ROM and save data and are written to a Nintendo 3DS Game Card. The packaged software sold in stores is one example of card-based software.

The following table shows where an application is saved, how large it is, and how it handles save data, extended save data, and e-manuals.

**Table 2-1. Characteristics of Card-Based Software**

| Item | Description |
|---|---|
| Save location | Nintendo 3DS Game Card |
| Size | Card1: Between 128 MB and 4 GB. (The actual usable size is different.)<br>Card2: Between 512 MB and 2 GB. (Shared with save data.) |
| Save location | Backup region of a Game Card. (Differs from the ROM region.) |
| Size | Card1: Between 128 KB and 512 KB. (The actual usable size is different.)<br>Card2: Specifiable in units of 1 MB (up to half the capacity of the card). |
| Extended Save Data | Yes. (Extended save data can be deleted from the Extra Data Management screen.) |
| E-manual support | Required |

**Warning:**Contact Nintendo at support@noa.com before using Card2.

## 2.2. Downloadable Applications

These applications are imported to an SD card. Software that is purchased in Nintendo eShop and software that is downloaded via SpotPass are two examples of downloadable applications.

The following table shows where an application is saved, how large it is, and how it handles save data, extended save data, and e-manuals.

**Table 2-2. Characteristics of Downloadable Applications**

| Item | Description |
|---|---|
| Save location | SD card |
| Size | A card application that supports download sales can be up to 4 GB.<br>If an application is only available for sale by download, it can generally be up to 512 MB.<br>(Contact Nintendo support at support@noa.com if you require a size greater than 512 MB.) |
| Save location | SD card. (A save data region is allocated when the application is imported. This region is deleted along with the downloadable application. You cannot delete the save data alone.) |
| Size | Between128 KB and 512 KB. (The actual usable size is different.) If a larger size is required, contact Nintendo support at support@noa.com. |
| Extended Save Data | Yes. (Extended save data is not deleted along with the application. However, it can be deleted from the Extra Data Management screen.) |
| E-manual support | Required |

**Warning:** If you submit a master submission as a downloadable application (CIA file), you will not be able to request sales as a card-based application later. If you are considering dual distribution, submit the master submission as a card-based application (CCI file).

## 2.2.1. Importing Downloadable Applications

To run a downloadable application, you must first import it to an SD card.

The retail version of the application is imported after it has been completely downloaded from Nintendo eShop or elsewhere.

You can use either the DevMenu or the PARTNER-CTR software debugger to import an application during development.

To import an application with the DevMenu, you must first write its CIA file to an SD card.

To import an application with the PARTNER-CTR software, either load the CIA file (you can even use a drag-and-drop operation) or run the IMPORT command.

To import an application with the PARTNER-CTR software, either load the CIA file (you can even use a drag-and-drop operation) or run the IMPORT command.

**Note:** For more information about how to use DevMenu, see the DevMenu page in the CTR Tools section of the CTR-SDK documentation.

For more information about the PARTNER-CTR commands, see the *PARTNER-CTR Debugger Manual*.

For more information about the IS-CTR-DEBUGGER commands, see the *IS-CTR-DEBUGGER Help*.

### 2.2.1.1. Importing and Overwriting Downloadable Applications

After a downloadable application is imported, it is identified by its unique ID. After you have imported a downloadable application, you cannot import another downloadable application with the same unique ID unless it has a newer version, in which case the new application overwrites the existing one.

Save data is normally preserved, even when an imported application overwrites an existing application. To prevent bugs from occurring, you must manage save data versions and other such information carefully. If the save data region for the application being imported is not the same size as the save data region for the application that has already been imported, the entire save data region will be cleared.

**Warning:** You are prohibited from resizing save data when you change the retail version.

Use the DevMenu or the Software Management screen under System Settings to delete downloadable applications that have been imported, when the save data region needs to be cleared, or before importing an earlier version of a downloadable application for debugging. Note that extended save data is not deleted along with a downloadable application.

## 2.2.2. Advance Download

An advance download function is used to enable a downloadable application to be purchased and downloaded from Nintendo eShop before its release date.

See the chapter about Advance Download in the *3DS Overview*.

## 2.2.3. Demos

A demo can be developed as a downloadable application.

The retail and demo versions of a downloadable application are imported as separate applications, even if they have the same unique ID. This means that you cannot transfer the demo's save data itself to the retail application, although you can share extended save data.

Use caution when sharing extended save data, though. If the demo is started after starting the retail version of your application, the demo uses the data that was updated by the retail application.

A demo's save data is deleted along with the demo, but extended save data is not.

**Note:** For more information about the demo version, see *Nintendo 3DS Demo Version Creation Instructions – Download Format*.

### 2.2.4. Differences From Card-Based Software, WiiWare, and Nintendo DSiWare

See the chapter about downloadable applications in the *3DS Overview*.

### 2.2.5. Importing TWL NAND Applications

You can use the `–srl` option for `ctr_makecia` to create a CTR CIA file from an SRL file that was built as a TWL NAND application. This CIA file can be imported to an SD card just like a downloadable application.

For more information, see the reference for the `ctr_makecia` tool in the CTR-SDK documentation.

**Sample Usage:**

```
ctr_makecia32.exe -srl MyTwlApp.srl -o MyTwlApp.cia
```

**Warning:** Use a CIA file created from an SRL file to check the operations of a Nintendo DSiWare title on a 3DS system. After you have imported your application, you can use the PARTNER-CTR software to run it but not to debug it.

## 2.3. Client Programs

Host devices distribute these applications to client devices via Download Play.

The following table shows where an application is saved, how large it is, and how it handles save data, extended save data, and e-manuals.

**Table 2-3. Characteristics of Client Programs**

| Item | Description |
|---|---|
| Save location | System NAND memory. (Only a single program is saved at a time. Whenever a program is received, it overwrites the previous one. The same version of the same program is not re-downloaded.) |
| Size | Up to 32 MB (uncompressed). |

| | |
|---|---|
| Save data | No |
| Extended save data | Yes. (Extended save data is not deleted when a client program is overwritten. However, it can be deleted from the Extra Data Management screen.) |
| E-manual support | Not Required |
| Comments | Client programs are not displayed in the HOME Menu. The banner (model and sound) is not required. |

**Note:** The `configtool` can be used to force the same version of the same program to be downloaded while debugging Download Play.

**Warning:** Although a downloadable application can become a Download Play server, at this time it cannot start a system update on connected client devices. When a downloadable application is running on a host device, it silently rejects connections from any client device that needs a system update. Applications must display a message provided by Nintendo before they start Download Play.

For more information, see the *CTR-SDK API Reference* for the DLP library.

CONFIDENTIAL

# 3. Implementation Tips

This chapter provides cautions and other helpful information related to developing an application.

## 3.1. Initialization Before Starting an Application

As explained in the *CTR System Programming Guide*, applications can implement their own versions of the following initialization processes that are run when an application is started.

- `nninitStartUp()` function definition
- C++ static initializers
- C static initializers

The `nninitStartUp()` function definition initializes memory-management features, and does so before the static initializers. This function is defined by a weak symbol. Applications can override it

by defining a function with the same name.

The default implementation can be found in the `nninitStartUpDefault()` function in `$CTRSDK_ROOT/sources/libraries/init/init_Default.cpp`. Nintendo strongly recommends that you replace this function with your own implementation of the `nninitStartup()` function.

## 3.2. Entry Function

The `nnMain()` function is the entry (main) function of a program, and is called after initialization. This function implements the application process.

When the `nnMain()` function exits, the application closes and control returns to the HOME Menu (or the development menu).

### 3.2.1. Handling Special Actions

Applications must handle the following actions.

- HOME Button presses.
- Transitions to Sleep Mode when the system is closed.
- POWER Button presses.

For more information, see the *3DS Programming Manual: System*.



# 4. Preparing to Build

This chapter explains how to create the files that you need to prepare before you build your application (separate from programming-related concerns).

The following files must be prepared before the application is built.

- RSF file
- BSF file
- E-manual

# 4.1. RSF Files

An RSF file is a settings file required to create CCI files and CXI files with `ctr_makerom`.

For more information, see the reference for the `ctr_makerom` tool in the CTR-SDK documentation.

## 4.1.1. Required Settings

The following settings are required in order to create an application.

**Table 4-1. Required Settings to Create an Application**

| Section | Subsection | Setting |
|---|---|---|
| TitleInfo | Category | Specify the value that corresponds to your application's type below.<br>• For card-based software: `Application`<br>• For downloadable applications: `Application` (For downloadable demos: `Demo`)<br>• For client programs: `DlpChild`<br>• For add-on content: `AddOnContents`<br>If you omit this setting, it is determined by the application type that is set by the DESC file when you build the application. |
| | UniqueId | Specify the unique ID that was issued to you by Nintendo.<br>Although you can specify a testing code (`0xFF000–0xFF3FF`) when you are testing and experimenting with your software, do not use the same ID for multiple applications.<br>You must specify the same unique ID for hosts and clients to support Download Play. |
| | ChildIndex | When you create a client program to be distributed through Download Play, specify its index here. |
| | DemoIndex | When you create a downloadable application demo version, specify its index (in the range from 1 to 255). For more information, see *Nintendo 3DS Demo Version Creation Instructions – Download Format*. |
| | Variation | When you create add-on content, specify its index (in the range from 0 to 255). |
| Rom | HostRoot | Specify the directory that stores the files to include in your ROM.<br>The specified directory is stored in the ROM archive as its root directory, along with all of the files and directories under it.<br>Specify `$(ROMFS_ROOT)` to access the `ROMFS_ROOT` of the OMakefile. |
| | Reject | Specify the files and directories to exclude from your ROM.<br>You can specify files with wildcard characters or regular expressions. |
| | SaveDataSize | Specify the size of the save data area in `nKB`/`nMB`/`nGB` format.<br>Specify `0KB` to omit a save data area. |
| CardInfo | MediaType | When you create a card application, specify the game card type (`Card1`/`Card2`).<br>Specify `Card2` for downloadable applications whose save data size |

| | | |
|---|---|---|
| | | exceeds 512 KB.<br>When omitted, the default is `Card1`. |

**Code 4-1. Sample RSF File (Required Settings)**

```
TitleInfo:
  Category:  Application
  UniqueId:  0xFF3FF
Rom:
  HostRoot:  $(ROMFS_ROOT)
```

## 4.1.2. Using Save Data

When your application uses save data, different items must be specified depending on the application type.

**Note:** Applications cannot actually use the full size specified for the save data region, because a part of it is used as an internal buffer for the file system. For more information about how much of each save data region can actually be used, see the Worksheet for Calculating the Save Data File System Capacity in the *CTR-SDK API Reference*.

You can also call the `nn::fs::GetArchiveFreeBytes()` function to get the amount of free archive space while your application is running.

### 4.1.2.1. Card Applications Using Card1 as the Game Card Type

Use the following settings when using this game card type.

**Table 4-2. Settings for Applications That Use Save Data (Card Applications Using Card1)**

| Section | Subsection | Setting |
|---|---|---|
| Rom | SaveDataSize | Specify `0KB`, `128KB`, or `512KB`. |

**Code 4-2. Sample RSF File (Using Save Data With a Card1 Card Application)**

```
Rom:
  SaveDataSize:  512KB
```

### 4.1.2.2. Downloadable Application

Use the following settings when using this game card type.

**Table 4-3. Settings for Applications That Use Save Data (Downloadable Application)**

| Section | Subsection | Setting |
|---|---|---|
| Rom | SaveDataSize | It differs depending on the MediaType specified.<br>Card1: Specify 0KB, 128KB, or 512KB.<br>Card2: Specify in the format of nMB or nGB.<br>Contact support@noa.com in advance if exceeding 50 MB.<br>In addition, we recommend a setting of up to 512 KB for dedicated sales of downloadable applications and up to 20 MB for dual distribution with card applications. |

**Code 4-3. Sample RSF File (Using Save Data With a Downloadable Application)**

```
Rom:
  SaveDataSize:  512KB
```

## 4.1.2.3. Card Applications Using Card2 as the Game Card Type

Use the following settings when using this game card type.

**Table 4-4. Settings for Applications That Use Save Data (Card Applications Using Card2)**

| Section | Subsection | Setting |
|---|---|---|
| Rom | SaveDataSize | Specify in the nMB or nGB format.<br>This value must be entered only in units of 1 MB. In addition, the physical limit is half the value specified by BasicInfo / MediaSize. |

**Code 4-4. Sample RSF File (Using Save Data With a Card2 Card Application)**

```
Rom:
  SaveDataSize:  32MB
```

## 4.1.2.4. Accessing the Save Data of Demo Versions

Use the following settings for applications that need to access save data created with a demo version.

**Table 4-5. Settings When Accessing the Save Data of Demo Versions**

| Section | Subsection | Setting |
|---|---|---|

| | | |
|---|---|---|
| AccessControlInfo | UseOtherVariationSaveData | Specify `true` or `false`.<br>Specify `true` to mount the demo version save data with `nn::fs::MountDemoSaveData`.<br>If omitted, the default is `false`. |

**Note:** When specifying the `AccessibleSaveDataIds` value of `AccessControlInfo`, specifying `true` for `UseOtherVariationSaveData` is interpreted more broadly to allow access to application save data and extended save data, even if the application's unique ID is not included.

### 4.1.2.5. Accessing the Save Data of Other Applications

Use the following settings for applications that need to access the save data of other applications.

**Table 4-6. Settings When Accessing the Save Data of Other Applications**

| Section | Subsection | Setting |
|---|---|---|
| AccessControlInfo | AccessibleSaveDataIds | Specify the save data unique ID and the extended save data ID that you want to access in the list of unique IDs. You can specify a maximum of six.<br>You can access save data (including demo versions) with a unique ID that matches the specified ID and extended save data with an extended save data ID that matches the specified ID.<br>Using these settings results in a broader interpretation of `UseOtherVariationSaveData`. For more information, see 4.1.2.4. Accessing the Save Data of Demo Versions. |
| | OtherUserSaveDataId1<br>OtherUserSaveDataId2<br>OtherUserSaveDataId3 | You can specify up to three unique IDs for save data to access. For save data with the same unique ID as the application, specify `true` for `UseOtherVariationSaveData` to use this without having to include the save data unique ID in these settings.<br>These settings cannot be used when specifying `AccessibleSaveDataIds`. |

## 4.1.3. Using Extended Save Data

Configure the following settings when your application uses extended save data.

**Table 4-7. Settings for Applications That Use Extended Save Data**

| | | |
|---|---|---|

| Section | Subsection | Setting |
| --- | --- | --- |
| AccessControlInfo | AccessibleSaveDataIds | Specify the save data unique ID and the extended save data ID that you want to access in the list of unique IDs. You can specify a maximum of six.<br>You can access save data (including demo versions) with a unique ID that matches the specified ID and extended save data with an extended save data ID that matches the specified ID.<br>If the application unique ID is included in these settings, you can access the extended save data regardless of the value specified for `UseOtherVariationSaveData`.<br>Using these settings results in a broader interpretation of `UseOtherVariationSaveData`. For more information, see 4.1.2.4. Accessing the Save Data of Demo Versions. |
| AccessControlInfo | UseExtSaveData | Specify `true`.<br>These settings cannot be used when specifying `AccessibleSaveDataIds`. |
| | ExtSaveDataNumber | You normally specify your application's unique ID.<br>If your application shares extended save data, specify the shared ID (the unique ID of the title with which data is shared).<br>For example, if your title is a sequel to another title with which it shares extended save data, specify the original title's unique ID.<br>These settings cannot be used when specifying `AccessibleSaveDataIds`. |

**Code 4-5. Sample RSF File (Extended Save Data)**

```
AccessControlInfo:
  AccessibleSaveDataIds:
    - 0xFF3FF
```

In the following cases, a new extended save data number is required that does not duplicate the unique ID of any application.

- Multiple applications have their own extended save data, and you want to use separate extended save data for sharing.
- A patch has been applied, and you want to add a new file to the extended save data, but the maximum number of files and directories that can be saved in the existing extended save data has been reached.
- You want to use multiple extended save data items with a single application independent of other applications for some other reason.

Contact Nintendo support at support@noa.com if you want to acquire a new extended save data number for one of these cases.

**Note:** If you specify `AccessibleSaveDataIds` of `AccessControlInfo`, the

UseExtSaveData and ExtSaveDataNumber specifications cannot be used. In this case, they can be used by including the extended save data ID of the extended save data to be accessed in AccessibleSaveDataIds, or by specifying true in UseOtherVariationSaveData if that extended save data ID and the application unique ID match.

**Warning:** The SD card can be accessed directly when the file system access attribute (FileSystemAccess in AccessControlInfo) is set to Debug. As a side effect, you can access data that cannot be accessed by a retail version, including save data and extended save data of other applications not specified in AccessibleSaveDataIds, save data from other applications not specified in OtherUserSaveDataID*, and extended save data not specified in ExtSaveDataNumber. Consequently, do not specify Debug while testing operations.

## 4.1.4. Settings for Creating a Master ROM to Submit

When creating a master ROM for submission, you must add the following settings or change the values that were specified when developing the application.

**Table 4-8. Settings for Creating a Master ROM to Submit**

| Section | Subsection | Setting |
|---|---|---|
| BasicInfo | Title | Specify the title of your application.<br>Specify $(TITLE) to access the TITLE of OMakefile. |
| | ProductCode | Specify the product code and downloadable content code issued to you by Nintendo. |
| | MediaSize | Specify the media size.<br>If MediaType is Card1, specify a value of 128MB, 256MB, 512MB, 1GB, 2GB, or 4GB.<br>If MediaType is Card2, specify a value of 512MB, 1GB, or 2GB.<br>For downloadable applications, specify the value that would be required if it were a card-based application. |
| | Logo | Specify the logo data type.<br>• Nintendo titles: Nintendo<br>• Titles for which Nintendo has acquired a publishing license: Distributed<br>• Licensee titles: Licensed |
| TitleInfo | UniqueId | Specify the unique ID that was issued to you by Nintendo.<br>If you were using a prototype code, check the value specified for ExtSaveDataNumber under AccessControlInfo. |

**Code 4-6. Sample RSF File (for Master ROM Submission)**

```
BasicInfo:
  Title:          $(TITLE)
  ProductCode:    "CTR-A-00001"
  MediaSize:      512MB
  Logo:           Licensed
TitleInfo:
  UniqueId:       0x41001
```

## 4.1.5. Remaster Version Settings

You must update the remaster version when you modify an application that has already been released.

You cannot update the remaster version in card-based software that has already been released, but you can update the version in downloadable applications that are being played by users.

**Table 4-9. Remaster Version Settings**

| Section | Subsection | Content |
|---|---|---|
| SystemControlInfo | RemasterVersion | Specify the remaster version of the CXI file. |

**Code 4-7. Sample RSF File (Remaster Version)**

```
SystemControlInfo:
  RemasterVersion:  1
```

This CXI remaster version is used unchanged as a downloadable application's version. A downloadable application's version is 1 if its CXI remaster version is 1. A downloadable application's version is incremented by one every time its CXI version is updated after the application has been released.

Save data is preserved when the version is updated. You must implement new versions of your downloadable application to appropriately handle save data from all previously released versions of the application. Handle this either by never changing the save data format, or by embedding the version number in the save data and automatically making any necessary data format changes when you upgrade.

Do not change the following RSF settings before and after you update your downloadable application's version.

**Table 4-10. Settings That You Must Not Change When You Update a Downloadable Application's Version**

| Section | Subsection | Content |
|---|---|---|
| BasicInfo | Title | Application title |
| | ProductCode | Product code |
| TitleInfo | UniqueId | Unique ID |

| Rom | SaveDataSize | Size of the save data region |
|---|---|---|

## 4.2. BSF Files

A BSF file is the settings file required to create ICN (icon data) and BNR (banner data) files with `ctr_makebanner`.

Use the BSF file to specify the icon file, model file, and sound file used to create both icon and banner data. You must create these files in advance.

Use CTPK files created by `ctr_TexturePackager` as the icon files to specify for `LittleIconFile` and `BigIconFile`.

Use a CBMD file created by `ctr_BannerModelConverter` as the model file to specify for `ModelFile`.

Use a BCWAV file created by `ctr_WaveConverter` as the sound file to specify for `SoundFile`.

A BSF file also has other application settings, such as the card region and title, which is displayed on the HOME Menu and elsewhere.

For more information, see the reference for the `ctr_makebanner` tool in the CTR-SDK documentation.

> **Note:** You do not need to prepare a model file and sound file for a client program, which does not require a BNR file.

**Code 4-8. Sample BSF File**

```
LittleIconFile:   resources/little.ctpk
BigIconFile:      resources/big.ctpk
ModelFile:        resources/model.cbmd
SoundFile:        resources/sound.bcwav

Region:           Japan
Ulcd:             True
AutoSave:         False
SaveData:         True
ThumbnailFrame:   0

RatingRequired:   True
CERO:             12

JPLongName:       Test application
JPShortName:      Test application
JPPublisher:      Test

ENLongName:       TestApplication
ENShortName:      TestApplication
ENPublisher:      Test
```

## 4.3. E-Manuals

Use CTR-ManualTools to create e-manual files. Note that an e-manual included in an application must have the filename `Manual.bcma`.

The size of the e-manual's binary file (BCMA file) is a maximum of 6656 KB per language.

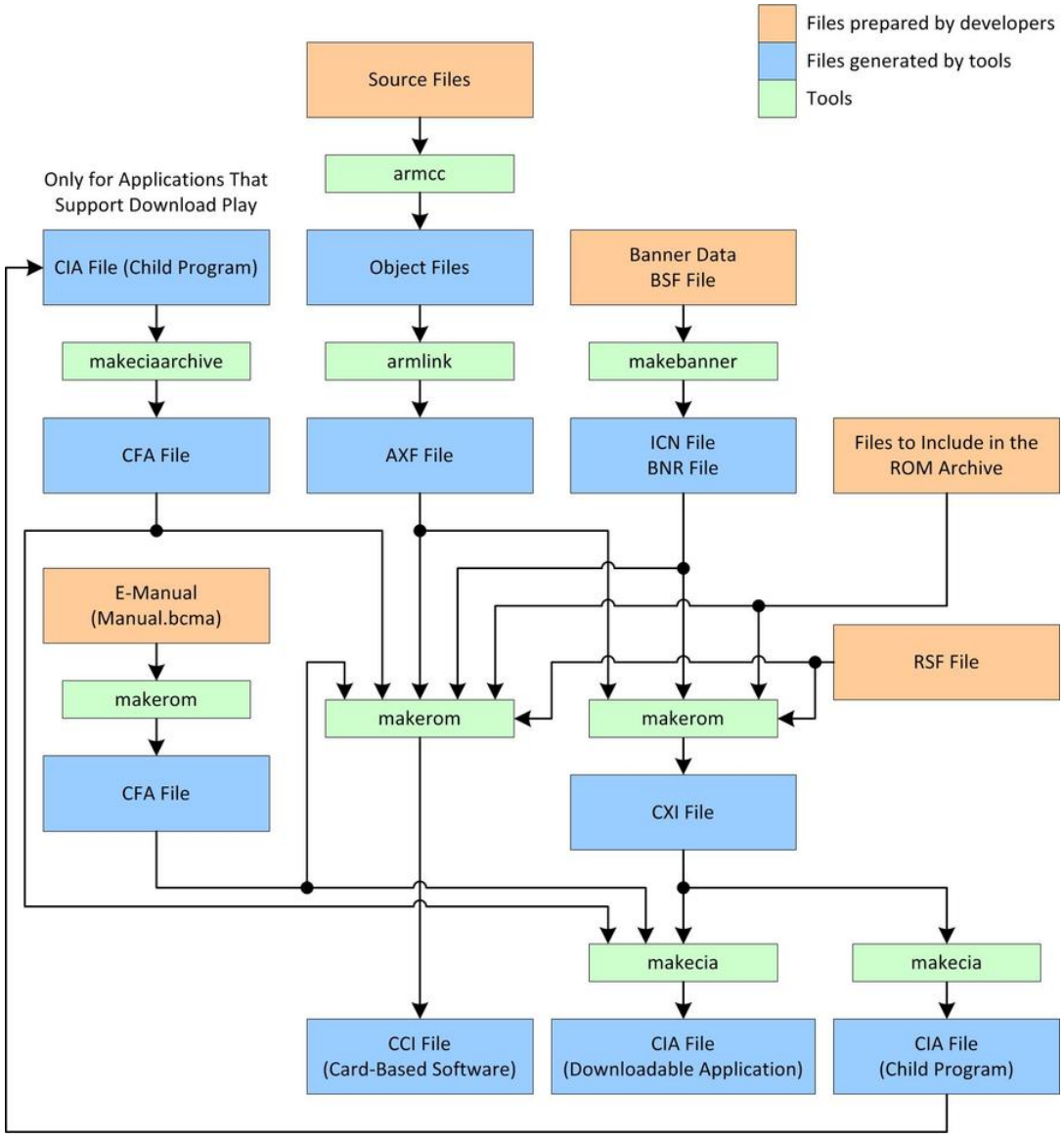
For more information, see the CTR-ManualTools documentation.

# 5. Building

The following figure provides an overview of the workflow for building an application.

For more information about the processing and files that are required to build an application, see the *CTR Guide to Developing a Build System*.

**Figure 5-1. Overview of the Steps for Building an Application**




# 5.1. Using the CTR-SDK Build System

If you are using the CTR-SDK's build system, you can configure environment variables, build options, and build variables in an OMakefile, and run `omake` to easily generate an executable file for your application. Simply change the value specified for `CTR_APPTYPE` and a few other settings to create a downloadable application, instead of card-based software.

If you have specified a BSF file specified with `CTR_BANNER_SPEC` in your OMakefile, the ICN and BNR files to specify for `CTR_BANNER` are generated based on the BSF file when you build your application. The `CTR_BANNER`, `CTR_ICON`, `CTR_NO_BANNER`, and `CTR_NO_ICON` values, even when set, are ignored when `CTR_BANNER_SPEC` is set.

For more information, see the Build Rules page in the CTR-SDK documentation.

## 5.1.1. Creating Card-Based Applications

The following table shows both required and recommended OMakefile settings for building card-based applications.

**Table 5-1. OMakefile Settings for Building Card-Based Applications**

| Setting | Description |
|---|---|
| CTR_APPTYPE | CARD |
| DESCRIPTOR | `$(CTRSDK_ROOT)/resources/specfiles/Application.desc` (standard applications)<br>`$(CTRSDK_ROOT)/resources/specfiles/ExtApplication.desc` (extended applications) |
| ROM_SPEC_FILE | Specify an RSF file. |
| CTR_BANNER | Specify a BNR file. |
| CTR_ICON | Specify an ICN file. |
| MANUAL_DIR | Specify a directory where files for an e-manual are stored. |
| CHILD_APPS[] | Specify one or more CIA files to distribute as client programs if your application supports Download Play. |

**Code 5-1. Sample OMakefile (for Card-Based Applications)**

```
SUPPORTED_TARGETS = CTR-TS.Process.MPCore.*

Omitted (header files, source files, libraries, shaders...)

TARGET_PROGRAM = MyApp
TITLE =          MyApp
CTR_APPTYPE =    CARD
DESCRIPTOR =     $(CTRSDK_ROOT)/resources/specfiles/Application.desc
ROM_SPEC_FILE =  MyApp.rsf
CTR_BANNER =     MyApp.bnr
CTR_ICON =       MyApp.icn
ROMFS_ROOT =     romfiles
MANUAL_DIR =     manual
CHILD_APPS[] =
    ../MyChildApp/images/$(BUILD_TARGET_DIR)/$(BUILD_TYPE_DIR)/MyChildApp.cia

include $(ROOT_OMAKE)/modulerules
build:  $(DEFAULT_TARGETS)
```

For settings not included in the table, see the Build Rules page in the CTR-SDK documentation.

## 5.1.2. Creating Downloadable Applications

The following table shows both required and recommended OMakefile settings for building downloadable applications.

**Table 5-2. OMakefile Settings for Building Downloadable Applications**

| Setting | Description |
|---|---|
| CTR_APPTYPE | SD |
| DESCRIPTOR | `$(CTRSDK_ROOT)/resources/specfiles/Application.desc` (standard applications, full version)<br>`$(CTRSDK_ROOT)/resources/specfiles/DemoVersion.desc` (standard applications, demo version)<br>`$(CTRSDK_ROOT)/resources/specfiles/ExtApplication.desc` (extended applications, full version)<br>`$(CTRSDK_ROOT)/resources/specfiles/ExtDemoVersion.desc` (standard applications, demo version) |
| ROM_SPEC_FILE | Specify an RSF file. |
| CTR_BANNER | Specify a BNR file. |
| CTR_ICON | Specify an ICN file. |
| MANUAL_DIR | Specify a directory where files for an e-manual are stored. |
| MANUAL_DISCLOSURE | Specify `public` to make e-manuals viewable for advance downloads. This directory does not need to be specified if the manual is not going to be viewable. |
| CHILD_APPS[] | Specify one or more CIA files to distribute as client programs if your application supports Download Play. |

**Code 5-2. Sample OMakefile (for Downloadable Applications)**

```
SUPPORTED_TARGETS = CTR-TS.Process.MPCore.*

Omitted (header files, source files, libraries, shaders...)

TARGET_PROGRAM = MyDownloadApp
TITLE =          MyApp
CTR_APPTYPE =    SD
DESCRIPTOR =     $(CTRSDK_ROOT)/resources/specfiles/Application.desc
ROM_SPEC_FILE =  MyDownloadApp.rsf
CTR_BANNER =     MyDownloadApp.bnr
CTR_ICON =       MyDownloadApp.icn
ROMFS_ROOT =     romfiles
MANUAL_DIR =     manual
MANUAL_DISCLOSURE = public
CHILD_APPS[] =
    ../MyChildApp/images/$(BUILD_TARGET_DIR)/$(BUILD_TYPE_DIR)/MyChildApp.cia

include $(ROOT_OMAKE)/modulerules
build:  $(DEFAULT_TARGETS)
```

For settings not included in the table, see the Build Rules page in the CTR-SDK documentation.

## 5.1.3. Creating Client Programs

The following table shows both required and recommended OMakefile settings for building client programs.

**Table 5-3. OMakefile Settings for Building Client Programs**

| Setting | Description |
|---|---|
| CTR_APPTYPE | NAND |
| DESCRIPTOR | `$(CTRSDK_ROOT)/resources/specfiles/DlpChild.desc` (standard applications)<br>**Note:** You cannot create client programs for extended applications. |
| ROM_SPEC_FILE | Specify an RSF file. |
| CTR_NO_BANNER | true |
| CTR_ICON | Specify an ICN file. |

**Code 5-3. Sample OMakefile (for Client Programs)**

```
SUPPORTED_TARGETS = CTR-TS.Process.MPCore.*

Omitted (header files, source files, libraries, shaders...)

TARGET_PROGRAM = MyChildApp
TITLE =          ChildApp
CTR_APPTYPE =    NAND
DESCRIPTOR =     $(CTRSDK_ROOT)/resources/specfiles/DlpChild.desc
ROM_SPEC_FILE =  MyChildApp.rsf
CTR_NO_BANNER =  true
CTR_ICON =       MyChildApp.icn
ROMFS_ROOT =     romfiles

include $(ROOT_OMAKE)/modulerules
build:  $(DEFAULT_TARGETS)
```

For settings not included in the table, see the Build Rules page in the CTR-SDK documentation.

# 5.2. Using Your Own Build System

This section describes the required steps to create an application when using your own build system instead of the build system provided with the CTR-SDK.

For more information, see *Guide to Developing a Build System* in the CTR-SDK documentation.

## 5.2.1. Creating AXF Files

Use `armlink` to link the object files and library files compiled with `armcc`, and create the AXF file.

**Sample Usage:**

```
armlink --datacompressor off --keep=nnMain \
--scatter=$CTRSDK_ROOT/resources/specfiles/linker/CTR.Process.MPCore.ldscript \
-o MyApp.axf MyApp.o MyAppLib.a ...
```

For more information, see the ARMCC documentation and *Guide to Developing a Build System* in the CTR-SDK documentation.

## 5.2.2. Creating ICN Files and BNR Files

Create the icon and banner files used in the application.

**Sample Usage:**

```
ctr_makebanner32.exe MyApp.bsf
```

If the only argument specified is a BSF file, the ICN file and BNR file are created with the same name as the BSF file, except for the different filename extensions. This example would create two files, `MyApp.icn` and `MyApp.bnr`.

For more information, see 4.2. BSF Files and the reference for the `ctr_makebanner` tool in the CTR-SDK documentation.

## 5.2.3. Creating E-Manual CFA Files

Run `ctr_makerom32.exe` with the following options to create the CFA file for the e-manual included with your application. You must first use CTR-ManualTools to create the `Manual.bcma` files for the e-manual.

- `-f data`
- `-rsf $CTRSDK_ROOT/resources/specfiles/Manual.rsf`
- `-DROMFS_ROOT=<`*`Directory where e-manual files (Manual.bcma) are stored`*`>`
- `-DMANUAL_DISCLOSURE=<true` to make e-manual viewable with advance download, or `false` if not>
- `-o <`*`Name of CFA file to create`*`>`

**Sample Usage:**

```
ctr_makerom32.exe -DROMFS_ROOT=manual_dir -DMANUAL_DISCLOSURE=true -f data \
-rsf $CTRSDK_ROOT/resources/specfiles/Manual.rsf -o Manual.cfa
```

For more information, see 4.3. E-Manuals and the reference for the `ctr_makerom` tool in the CTR-SDK documentation.

## 5.2.4. Creating Card-Based Applications

CCI files are the final ROM image format for card-based applications. These files are used for debugging or for ROMs to be submitted as masters.

### 5.2.4.1. Creating CCI Files

You need the following files to create a CCI file.

- AXF file
- DESC files (included in CTR-SDK)
- RSF file
- ICN files
- BNR files
- E-manual CFA files (when supporting an e-manual)
- Client program CFA files (when supporting Download Play)

Run `ctr_makerom32.exe` with the following options to create the CCI file.

- `-f card`
- `-rsf` *`<RSF file>`*
- `-desc $CTRSDK_ROOT/resources/specfiles/Application.desc`
  - For extended applications: `-desc $CTRSDK_ROOT/resources/specfiles/ExtApplication.desc`
- `-icon` *`<ICN file>`*
- `-banner` *`<BNR file>`*
- `-content` *`<CFA file>`*`:`*`<Index>`*
  - Specify 1 for the index value if the CFA file is for an e-manual.
  - Specify 2 for the index value if the CFA file is for a client program.
- `-o` *`<Name of CCI file to create>`*

For more information, see the reference for the `ctr_makerom` tool in the CTR-SDK documentation.

**Sample Usage:**

```
ctr_makerom32.exe MyApp.axf -f card -rsf MyApp.rsf \
-desc $CTRSDK_ROOT/resources/specfiles/Application.desc \
-icon MyApp.icn -banner MyApp.bnr \
-content Manual.cfa:1 -content MyChildApp.cfa:2 \
-o MyApp.cci
```

## 5.2.5. Creating Downloadable Applications

CIA files are the final ROM image format for downloadable applications. These files are used for debugging or for ROMs to be submitted as masters.

You must first create a CXI file before you can create a CIA file.

### 5.2.5.1. Creating CXI Files

You need the following files to create a CXI file.

- AXF file
- DESC files (included in CTR-SDK)
- RSF file
- ICN files
- BNR files

Run `ctr_makerom32.exe` with the following options to create the CXI file.

- `-f exec`
- `-rsf <RSF file>`
- `-desc $CTRSDK_ROOT/resources/specfiles/Application.desc`
  - For demo versions of standard applications: `-desc $CTRSDK_ROOT/resources/specfiles/DemoVersion.desc`
  - For extended applications: `-desc $CTRSDK_ROOT/resources/specfiles/ExtApplication.desc`
  - For demo versions of extended applications: `-desc $CTRSDK_ROOT/resources/specfiles/ExtDemoVersion.desc`
- `-icon <ICN file>`
- `-banner <BNR file>`
- `-o <Name of CXI file to create>`

For more information, see the reference for the `ctr_makerom` tool in the CTR-SDK documentation.

**Sample Usage:**

```
ctr_makerom32.exe MyDownloadApp.axf -f exec -rsf MyApp.rsf \
-desc $CTRSDK_ROOT/resources/specfiles/Application.desc \
-icon MyApp.icn -banner MyApp.bnr \
-o MyDownloadApp.cxi
```

### 5.2.5.2. Creating CIA Files

You need the following files to create a CIA file.

- CXI file
- E-manual CFA file
- Client program CFA files (when supporting Download Play)

Run `ctr_makecia32.exe` with the following options to create the CIA file.

- `-i <CXI file>`
- `-man <E-manual CFA file>`
- `-i <Client program CFA file>:2` (You must specify the index value.)
- `-o <Name of CIA file to create>`

For more information, see the reference for the `ctr_makecia` tool in the CTR-SDK documentation.

**Sample Usage:**

```
ctr_makecia32.exe -i MyDownloadApp.cxi -man Manual.cfa -o MyDownloadApp.cia
```

## 5.2.6. Creating Client Programs

Client programs included with card-based applications or downloadable applications are bundled into a single CFA file, as specified in the options when creating the card-based application or downloadable application.

You must first create the CIA file before you can create a CFA file, and to create the CIA file, you must first create a CXI file. In order, create the CXI file, create the CIA file, and then create the CFA file.

## 5.2.6.1. Creating CXI Files

You need the following files to create a CXI file. Banners are not displayed, so you do not need a BNR file.

- AXF file
- DESC files (included in CTR-SDK)
- RSF file
- ICN files

Run `ctr_makerom32.exe` with the following options to create the CXI file.

- `-f exec`
- `-rsf <RSF file>`
- `-desc $CTRSDK_ROOT/resources/specfiles/DlpChild.desc`
- `-icon <ICN file>`
- `-o <Name of CXI file to create>`

For more information, see the reference for the `ctr_makerom` tool in the CTR-SDK documentation.

**Sample Usage:**

```
ctr_makerom32.exe MyChildApp.axf -f exec -rsf MyChildApp.rsf \
-desc $CTRSDK_ROOT/resources/specfiles/DlpChild.desc \
-icon MyChildApp.icn \
-o MyChildApp.cxi
```

## 5.2.6.2. Creating CIA Files

You need the following files to create a CIA file.

- CXI file

Run `ctr_makecia32.exe` with the following options to create the CIA file.

- `-i <CXI file>`
- `-o <Name of CIA file to create>`

For more information, see the reference for the `ctr_makecia` tool in the CTR-SDK documentation.

**Sample Usage:**

```
ctr_makecia32.exe -i MyChildApp.cxi -o MyChildApp.cia
```

### 5.2.6.3. Creating CFA Files

You need the following files to create a CFA file.

- All CIA files for client programs to distribute

Run `ctr_makeciaarchive32.exe` with the following options to create the CFA file. Client programs distributed for Download Play are bundled into a single CFA file.

- `-cia <List of client program CIA files>`
- `-o <Name of CFA file to create>`

For more information, see the reference for the `ctr_makeciaarchive` tool in the CTR-SDK documentation.

**Sample Usage:**

```
ctr_makeciaarchive32.exe -cia MyChildApp.cia -o MyChildApp.cfa
```

This CFA file is used when creating a card-based application with the `-content` option (index 2), or when creating a downloadable application with the `-i` option (index 2).

# 6. Debugging

Use the following combinations of hardware and debugging software to debug.

**Table 6-1. Combinations of Hardware and Debugging Software to Use for Debugging**

| Hardware | Debugging Software | 3DS Application | SNAKE-Compatible Application |
|---|---|---|---|
| PARTNER-CTR Debugger | PARTNER-CTR | ○ | × |
| IS-SNAKE-BOX | IS-CTR-DEBUGGER | ○ | ○ |

# 6.1. Debugging Methods According to the Menu Displayed at Startup

You can switch the menu that is displayed when you start PARTNER-CTR Debugger or IS-SNAKE-BOX. To switch the menu, from Config Tools, select **Menu Settings** > **Menu**. How you debug applications differs depending on the menu that is displayed.

## 6.1.1. Starting Your Application From the Development Menu

For normal debugging, you load a CCI file or CIA file into PARTNER-CTR or IS-CTR-DEBUGGER. This is the recommended debugging method, as it allows you to start debugging right after starting.

However, card-based applications written to a CTR flash card cannot be debugged when started from the development menu, unless they are started in the same way as when starting from the HOME Menu. This is also the case when starting downloadable applications from the DevMenu.

## 6.1.2. Starting Your Application From the HOME Menu

Starting from the HOME Menu does not happen via the debugging software, so you cannot debug this way without carrying out additional steps.

Use the following procedure to debug applications started from the HOME Menu. (This procedure allows you to debug both card-based applications, and downloadable applications that have been imported into an SD card.)

### Using PARTNER-CTR

**Note:** Reset to have the HOME Menu display applications loaded into the PARTNER-CTR emulation memory.

1. Start PARTNER-CTR.

2. Start the application from the HOME Menu.
3. Run the ATTACHA command in the PARTNER-CTR command window.
   > ATTACHA
4. Run the LS command to load the symbol file.
   > LS MyApplicaton.axf

This results in debugging an application that has already started, so you cannot debug anything at start or immediately thereafter.

For more information about the PARTNER-CTR commands, see the *PARTNER-CTR Debugger Manual*.

#### Using IS-CTR-DEBUGGER

> **Note:** You can display on the HOME Menu even when loading a CCI file or CIA file in IS-CTR-DEBUGGER.

1. Start IS-CTR-DEBUGGER.
2. Start the application from the HOME Menu.
3. On the IS-CTR-DEBUGGER menu, select **Debug** > **Attach to Running Program**.
4. On the IS-CTR-DEBUGGER menu, select **File** > **Open** and load an AXF file.

This results in debugging an application that has already started, so you cannot debug anything at start or immediately thereafter.

## 6.2. Starting the CIA File With Debugger Software

When loading the CIA file directly with the debugging software, a C8804478 error may occur. This occurs when there is no CXI file in the same directory as the CIA file, and when the application is not using the default unique ID.

You must support either of the following if you want to run a CIA file directly from the debugging software.

- Building the application using the default unique ID
- Placing a CXI file in the same directory as the CIA file

# 7. Using DLLs (Dynamic Modules)

You can not only use DLLs to reduce your program's memory consumption, but also to shorten start times (during which the Nintendo 3DS logo is visible) because only a static module is loaded when the program starts.

## 7.1. Implementation

If DLLs are not used, you can implement your entire application in a single (main) program. If DLLs are used, you can split your implementation between several dynamic modules that are loaded into memory on demand. If you split your implementation into small dynamic modules that are too fine-grained, however, it will take a long time to load the modules and the frequent switching between modules could complicate memory management.

For more information, see *DLL Manual* in the CTR-SDK documentation and *3DS Programming Manual: System*.

## 7.2. Preparing to Build

Follow the same procedure described in 4. Preparing to Build.

## 7.3. Building

### 7.3.1. Using the CTR-SDK Build System

You can easily build an application that supports dynamic modules by simply paying attention to how your modules' source code is structured and what you put into your OMakefile. Note, however, that a separate build directory must be created for each module, distributing the individual source files and OMakefiles so that the application (static module) has the same parent directory as the dynamic modules.

The following settings have been added to the OMakefile to build dynamic modules.

**Table 7-1. Additional Settings for Building Dynamic Modules**

| Setting | Description |
|---|---|
| TARGET_MODULE | Specify the name of the dynamic module. |
| MODULE_LIST | Specify all static and dynamic modules except for this one. |
| MODULE_EXPORT_TYPE | Specify the type of data that is exported from the dynamic module.<br>• SYMBOL: Names (default)<br>• INDEX: Indices<br>• OFFSET: Offsets |
| MODULE_ENABLE_OFFSET | This has a default value of true, which requires all modules to be built at the same time. When this is false, you can build modules at any time, but you will no longer be able to specify offsets as the type of data to export. |
| MODULE_ADD_DEBUG_INFO | Specifies whether to include the CRO file's debug information along with the CRR file to be created. The default setting is "true" so source debugging can be performed by the debugger. If the setting is "false", source debugging cannot be performed. You cannot debug when this is set to false, but the size of the CRR file will be smaller. |

You do not need to specify an RSF, ICN, BNR, or DESC files when you build a dynamic module. Note that you specify a dynamic module's name in TARGET_MODULE instead of TARGET_PROGRAM. A CRO file is generated when you build a dynamic module.

**Code 7-1. Sample OMakefile for a Dynamic Module**

```
TARGET_MODULE = Module1
MODULE_LIST = ../App ../Module2
SOURCES = module.cpp

include $(ROOT_OMAKE)/modulerules
build:  $(DEFAULT_TARGETS)
```

To build a static module, simply add MODULE_LIST to the normal build settings. A CRS and CRR file are generated when you build a static module.

**Code 7-2. Sample Setting Added to an Application's OMakefile**

```
MODULE_LIST = ../Module1 ../Module2
```

A ROM archive must include the CRO, CRS, and CRR files that are generated. The CRO and CRS files do not need to be placed anywhere in particular, and they can be either compressed or uncompressed. The .crr directory that is generated directly under the root directory must only contain one uncompressed CRR file and nothing else.

With the following sample OMakefile, OMake will automatically store each file when you build an application that uses dynamic modules.

**Code 7-3. Sample OMakefile for Automatically Storing Files in a Dynamic Module**

```
MODULE_NAMES       = Module1 Module2
MODULES_ROOT       = ../

MODULE_CRO         = $(addprefix $(ROMFS_ROOT)/, $(addsuffix .cro,
$(MODULE_NAMES)))
STATIC_CRS         = $(ROMFS_ROOT)/static.crs
STATIC_CRR         = $(ROMFS_ROOT)/.crr/static.crr
ROMFS_DEPENDENCIES = $(MODULE_CRO) $`(STATIC_CRS) $`(STATIC_CRR)

foreach(TARGET, $(BUILDER.getTargets $(SUPPORTED_TARGETS)))
    foreach(module, $(MODULE_NAMES))
        src = $(file $(TARGET.getImageDirectoryOf
$(MODULES_ROOT)$(module))/$(module).cro)
        dst = $(addprefix $(ROMFS_ROOT)/, $(addsuffix .cro, $(module)))

        $(dst) : $(src)
            cp $< $@

        $(STATIC_CRS): $(TARGET.getImageDirectory false)/$(TARGET_PROGRAM).crs
            cp $< $@

        $(STATIC_CRR): $(TARGET.getImageDirectory false)/$(TARGET_PROGRAM).crr
            mkdir -p $(dirname $@)
            cp $< $@
```

For more information, see the *CTR-SDK Build System Manual (for DLLs)*.

### 7.3.2. Using Your Own Build System

To build dynamic modules without using the build system provided by the CTR-SDK, you must add partial linking, symbol resolution, and other special processing.

For more information, see *Guide to Developing a Build System (for DLLs)* in the CTR-SDK documentation.

## 7.4. Debugging

When building an application, source debugging can be performed using the debugger if the CRO file's debug information is included along with the CRR file.

For more information about the methods of including the CRO file's debug information with the CRR file, see 7.3.1. Using the CTR-SDK Build System and the *CTR-SDK Build System Manual (for DLLs)*.



# 8. Developing Card Applications That Can Be Sold

# via Download

Ever since Nintendo started developing the CTR system, one of its goals has been to enable all card applications to also be sold from Nintendo eShop, without modification. Nintendo designed the CTR-SDK to minimize the amount of separate code that must be written for card-based software and downloadable applications, and whenever possible, the requirements in the *Guidelines* are the same for both versions of the application.

This chapter describes the steps required to sell the same application both as a card-based software application and a downloadable application (this is called "dual distribution"). It describes the procedures for two different scenarios: when you are developing a new application for dual distribution, and when you have already developed a card application, and want to offer if for sale via download. It also describes the steps to take when you need to revise your application.

**Warning:** If you submit a master submission as a downloadable application (CIA file), you will not be able to request sales as a card-based application later. If you are considering dual distribution, submit the master submission as a card-based application (CCI file).

## 8.1. Developing a New Application for Dual Distribution

The procedures for developing a new card application for dual distribution as a downloadable version are essentially the same as for developing an ordinary card-based software application. This section describes aspects of the development process that require special attention.

### 8.1.1. The Planning Stage

Card-based software applications and downloadable applications for the same title are treated as the same product, with different sales channels. **Make sure that the card and downloadable applications work the same.**

**Note:** Contact Nintendo support at support@noa.com if you want your card applications and downloadable applications to work differently.

If Nintendo approves differences, note the specific differences in Checksheet Editor in the Program Specifications checksheet, under "Nintendo-Approved Specifications."

## 8.1.2. Getting the Product Code and Unique ID

A dual-distribution application uses the same product code and unique ID for both the card and downloadable versions. **Do not get separate product codes or unique IDs for the package version and downloadable version.**

> Examples: For example, if you have obtained the product code `CTR–P–AAAJ(JPN)` for your card application, you would use `CTR–P–AAAJ` for the downloadable application's product code. You would **not** use `CTR–N–AAAJ`, which would indicate that the product is a downloadable application.

You actually manage your product using the product code for the downloadable version—in this example, `CTR–N–AAAJ`. When you produce the software or submit the master ROM, however, it is fine to consider these to be the same product code and unique ID.

## 8.1.3. E-Manual

**You must include the same e-manual with your card and downloadable applications.**

**To create your e-manual, use the latest version of the *CTR Electronic Manual Creation Guide, Part 2 – Template Units*.**

## 8.1.4. Save Data and Extra Data

If a system has both the package and downloadable versions of an application, each application will access separate save data (the package version will access the backup memory on the Game Card, and the downloadable version will access the SD card), but both will access the same extra data (extended save data). Consider the following when deciding how your application will deal with this situation.

This situation will also occur if a user plays multiple Game Cards for the same title on a single system. Regardless of whether you support download sales, a card application should be able to handle this situation without any fatal bugs occurring. And if your application can support multiple Game Cards, it should be able to support download sales through some additional handling.

Your application need not offer special handling for such cases (such as enabling users to migrate their save data if they own the package version and later purchase the downloadable version), but if you implement a feature to support multiple Game Cards (such as data migration), confirm whether this feature can be repurposed to handle the situation of a package version and downloadable version co-existing.

## 8.1.5. Save-Data Rollbacks

Although users cannot roll back the save data of a card application to an earlier state, it is easy to do so for a downloadable application using a personal computer. (Users can back up the contents of the entire SD card to their PC at the desired save point, and then roll back the save data by copying the PC backup to the SD card.)

Applications can detect save-data rollbacks using the following functions from CTR-SDK 4.2.2 and later: `SetSaveDataSecureValue()` and `VerifySaveDataSecureValue()` (or `VerifySaveDataSecureValueLoosely()`). When called from a card application, the `SetSaveDataSecureValue()` function will have no effect, and the `VerifySaveDataSecureValue()` and `VerifySaveDataSecureValueLoosely()` functions will always return `true`. You can also safely use these functions in a card application that supports download sales, because they have no effect if your application is running as a card application.

When planning your game, consider using these functions if rollbacks of save data would have a fatal effect on game balance or other factors. If you create game-ending penalties for users (such as making all save data unusable in the event of a rollback), cases such as the following may occur. When deciding on your application's specifications, also fully consider how you will communicate this to your users, and how you will provide support.

- The user may create a backup of the save data on a PC without knowing about the penalty for rollbacks, and delete the data recorded to the SD card in the last call to the `SetSaveDataSecureValue()` function.
  For downloadable games that do not take measures against rollbacks, there is a risk that users will lose their data without knowing that they are doing something that is not allowed.

- If the system is repaired, support technicians may be able to rescue the save data on the SD card, but not in the system NAND memory.
  The value stored using the `SetSaveDataSecureValue()` function will be lost, and become inconsistent with the value in the remaining save data. In this situation, the penalty will be incurred if you are using the `VerifySaveDataSecureValue()` function, but if you are using the `VerifySaveDataSecureValueLoosely()` function, the user can use the save data on the SD card before repair, because this function will return TRUE in this instance. However, the save data on the SD card before repair might have been rolled back, so you must decide whether your business model can allow this.

## 8.1.6. In-Game Notations and Displays

The text messages, copyright notations, and other content in the game must not pose problems on either the card or downloadable application version.

The following are examples of in-game messages where you must take into account the differences between card and downloadable applications when drafting effective messages.

- Different storage media
  Card-based software applications store the software and save data to a Game Card.
  Downloadable applications store the software and save data on an SD card.

Sample good and bad in-game messages:
Bad: "Saving. Do not remove the Game Card."
Good: "Saving. Do not remove the Game Card or SD card."
Good: "Saving. Do not remove the card."

- Inclusion of printed materials
  Card applications come with printed material, such as a quick start guide, but downloadable applications do not.

  Following are good and bad examples of in-game messages.
  Bad: "See the included quick start guide for instructions on game controls."
  Good: "See the e-manual for instructions on game controls."
  Good: "See the in-game Help for instructions on game controls."

Use the following language in user-facing text if you want to communicate information to users differently depending on whether they are using a card or downloadable application.

Downloadable application → Downloadable version

Card application → Package version

Use user-friendly terminology as in the examples.

## 8.1.7. Download Play

If the parent system is using a card application for Download Play on the 3DS, a system update is performed via local communication (DUP) during the Download-Play sequence. However, the specifications do not allow the use of DUP if the parent system is using a downloadable application. For this reason, the child system must perform a system update beforehand if its firmware version is earlier than the required version.

**Do one of the following** to communicate this effectively to your users.

- Display the following message on the parent system for Download Play, either while or before accepting child systems:
  "If the message "System update required" appears on the other system, Download Play will not be available.
  Have the other player press **Cancel**, and update the system from System Settings. Selecting "OK" will not update the system."

  If This message is too long to display, you must, at a minimum, inform the user that Download Play is not available on old system versions.
  Only display this message if the argument *pNotice* is set to TRUE in the function `nn::dlp::Server(WithName)::Initialize`. The parameter *pNotice* is set to FALSE when either a DUP is available because the parent system is using the card application, or a

future system update has made DUP available for downloadable applications as well.

- Explain in the e-manual that a system update is required for child systems.
  Sample user-facing text can be found in *CTR Electronic Manual Creation Guide, Part 2 – Template Units.*

## 8.1.8. Media Access Speed

The *File System* guidelines document recommends that implementations generally not depend on the access speed of recording media, but note that ROM and save-data access speeds differ depending on whether an application is run as a card application or a downloadable application.

Although there is no significant difference in the read speeds of ROM data between Game Cards and SD cards, data reads and writes from/to an SD card by a downloadable application are much faster than when using the backup memory of the Game Card.

> **Note:** For more information about access speeds, see *3DS Performance Tips*.

When accessing data from the ROM and extra data (on the SD card) simultaneously, a card application will be accessing two different media, but a downloadable application will be accessing the SD card for both. This could cause a difference in processing time for the same action, depending on whether it is a card or downloadable application.

You can set the priority of file access (see Access Priority Setting in *3DS Programming Manual: System*). For example, if you do not set a high priority for streaming data from an SD card, this may not cause any problems for a card application, but result in streaming delays in the downloadable application.

## 8.1.9. Generating (Building) Application Binaries

For your master ROM submission, you only submit a CCI file (binary for card applications). Build your application in the same way as you have done with card applications previously.

To test the application as a downloadable application, convert the CCI file into a CIA file, and import it onto an SD card.

> **Note:** To convert a CCI file into a CIA file, run `makecia` with the `cci` option.

Use the Master Editor tool to make sure that the CIA file was generated correctly from the CCI file. Specifically, look on the CXI and CFA tabs for the CCI and CIA files (all of them if there are more

than one), and under "Hash value (CRC32)," make sure that the value for "Content total" is the same for both.

## 8.1.10. Debugging

Debug both your card application and downloadable application. You do not need to perform twice as much debugging as you have performed previously. Just devote a certain portion of your debugging on the system to the downloadable application.

If you test running both a card and downloadable application on a single system, perform the same tests as when you test running multiple card applications on a single system.

Debug using the SD card bundled with the development hardware or a genuine Nintendo SD card.

> **Note:** See Media Access Speeds in the *File System* guidelines document.

### 8.1.10.1. Reusing Save Data

You can use the `SaveDataFiler` tool (included in CTR-SDK) to write a card application's save data to an SD card, and then load it from the downloadable version.

## 8.1.11. Master ROM Submission

The procedure for submitting a master ROM for a card application that will now be sold as a downloadable application is the same as for a standard card application. Submit the same master data (CCI file) for the card and downloadable applications. You do not need to submit a separate CIA file for download sales, nor do you need to create separate packages for submitting the package version and downloadable version.

Use CTR Master Editor to submit your master data. Under "Application purpose," select "General sales," and select both the "Card sales" and "Download sales" check boxes **to show explicitly that you support download sales**. **This submits your application as a dual-distribution application, but contact Nintendo support at support@noa.com beforehand if you want to start download sales immediately, before the cards go into production.**

If you want to treat card applications and downloadable applications differently (with special approval from Nintendo), **describe the specific differences in "Nintendo-Approved Specifications" on the Program Specifications checksheet in Checksheet Editor.**

# 8.2. Selling a Previously Launched Card Application as a Downloadable Application

In most cases, making a previously launched card application support download sales only requires adding an e-manual. This section describes aspects of the development process that require special attention.

## 8.2.1. SDK Version

If you have a master ROM that you submitted as a card application, and you want to use it as a downloadable application, or if the only change is an added or modified e-manual, the rules listed under the CTR-SDK Versions Accepted for Master Submission page on WarioWorld currently do not apply. In other words, it is fine to use the same SDK version that you were using.

Contact Nintendo support at support@noa.com if you need to rebuild your application due to some change. However, it is fine to keep the current SDK version for the following types of change:

- Minor revisions, such as changing some of the messages or UI displays.
- Remaster to include content in a previously released patch.

If you rebuild a modified version of your application, **you must submit a revision history in the applicable checksheet of Checksheet Editor or a separate document that describes in detail each of the changes from the previous version that passed Lotcheck.** See the *CTR Master ROM Submission Guidelines* for details.

## 8.2.2. E-Manual

In general, **you must support the items that require immediate support as spelled out in the latest version of the *CTR Electronic Manual Creation Guide, Part 2 – Template Units***. Create or revise your e-manual as required, and add it to your previously released binary (replacing any existing e-manual) using CTR-ROM Editor. **If adding the e-manual causes the ROM to exceed the maximum size available to the application, also change the media size.**

**Table 8-1. Need for New or Revised E-Manuals**

| E-Manual | Necessary Changes |
|---|---|
| None | Create a new e-manual and submit it. |
| Yes | Total replacement is not necessary, regardless of the transition period. You just need to modify any parts that require immediate support subsequent to the version of the Creation |

| | |
|---|---|
| | Guide that was in effect at the time of the product launch. |

**Note:** Currently, the *E-Manual Production* manual is bundled with the *CTR E-Manual Template* collection, and the name has been changed to the *E-Manual Production* manual.

## 8.2.3. Save Data Compatibility

When revising applications that support dual distribution, if there are multiple versions on the market as a result of revisions to the card-applications side of distribution, consider the following.

When there is no save data compatibility between any version of the card application on sale and the distributed downloadable application, after the save data has been migrated with the Move Save Data tool, a fatal error results and the save data is unusable. If this situation applies, contact support@noa.com. Nintendo will work to ensure that relevant save data is not migrated by the Move Save Data tool.

## 8.2.4. Save-Data Rollbacks

As described in 8.1.5 Save-Data Rollbacks, users of downloadable applications can roll back their save data by using a PC to manipulate their SD cards—something that was not possible with card applications. You must verify during the planning stages whether rolling back save data will disrupt the game balance.

## 8.2.5. In-Game Notations and Displays

Check whether the text messages, copyright notations, and other content in the game pose problems for downloadable application sales.

**Note:** If the application has not changed, Nintendo will not re-check the messages, notations, displays, and other content for problems.

## 8.2.6. Download Play

8.1.7 Download Play describes the operations of Download Play. **If your application supports Download Play and does not display the messages set forth in 8.1.7 Download Play, you must**

**place them in the e-manual.**

## 8.2.7. Testing the Application

You must also test your application for bugs, even if you run it from an SD card as a downloadable application.

Also pay attention to the information in 8.1.4. Save Data and Extra Data, 8.1.8. Media Access Speed, and 8.1.9. Generating (Building) Application Binaries.

## 8.2.8. Master ROM Submission

Resubmit a CCI file with an attached e-manual, in accordance with the *CTR Master ROM Submission Guidelines*.

Use CTR Master Editor to submit your master data. Under "Application purpose," select "General sales," and select both the "Card sales" and "Download sales" check boxes **to show explicitly that you support download sales**.

The following table shows the required deliverables and their versions.

**Table 8-2. Required Deliverables and Versions for Submitting Master ROMs**

| | Submission Items | | | | Version | |
|---|---|---|---|---|---|---|
| | Master ROM Submission Sheet | Master ROM | Checksheet | Revision History | Remastered Version | Submission Version |
| Master ROM data identical to data submitted previously | Required | Not Required | Not Required* | Not Required | Same as last time | Same as last time |
| Only added or replaced e-manual | Required | Required | Not Required* | Required | Same as last time | Previous + 1 |
| Application has been changed | Required | Required | Required | Required | Previous + 1 | 0 |

* If the card application and downloadable application differ, and you did not note this difference in the previous submission documentation, describe the specific differences under "Nintendo-Approved Specifications" on the Program Specifications checksheet in Checksheet Editor. If any of the following applies to your title, and you did not note this in the previous submission documentation, you must resubmit your checksheet.

- Titles that use the save-data rollback prevention support function
- Titles that access the save data of other applications

**Note:** In general, use the latest version of the Master Editor and Checksheet Editor to create your deliverables.

However, using the newest version of Master Editor can cause the following errors. Ignore this error if you have not yet rebuilt to include fixes.

- Settings with values configured and settings with no values configured can get mixed up for the long title, short title, and publisher name settings.

## 8.3. Revising Your Application

This section describes the steps required when you revise an application marketed via dual distribution as a package and downloadable version.

We recommend that you use a patch to make revisions. See the chapter about patches in the *3DS Overview*.

**Warning:** When revising a title that uses a fake client, special consideration is required if a fake client that differs from the remaster version will connect to the DLP server.

See the chapter about fake client features in the *Patch Manual*.

### 8.3.1. Revising Your Application After Launch

**As a general rule, create and submit a patch for any revisions.** There is no need to submit revised versions.

Only submit a revised version if you plan on producing a revised version with a new Game Cards with the revisions applied. In this case, select only the card sales option under the "Application purpose" section of the Master ROM Submission Sheet. This master ROM will only be used to manufacture Game Cards. It will not be distributed through Nintendo eShop.

Patches work for the majority of cases where a fix is needed. In such cases where a patch cannot be applied, submit the updated version for dual distribution. In this case, in the Master Editor tool, in **Application purpose**, select both **Card sales** and **Download sales**.

**Note:** E-manual revisions and fixes for Download Play child programs are also applicable with

patches.

### 8.3.2. Distributing a Card Application That Has Already Been Released and Patched

It is fine to make the unpatched version of the application (the content that was released on the card) ready for dual distribution in accordance with 8.2. Selling a Previously Launched Card Application as a Downloadable Application, and then resubmit it. However, you must confirm that the downloadable application runs correctly when it is patched.

You can also submit a revised version.



# Revision History

## Version 1.3 (2015-11-05)

### Changes

- 2.2. Downloadable Applications
    - Added a note that you cannot change a downloadable application to card-based application sales later.
- 8. Developing Card Applications That Can Be Sold via Download
    - Added a note that you cannot change a downloadable application to card-based application sales later.
- 8.3. Revising Your Application
    - Added a note about revising titles using fake clients.

# Version 1.2 (2015-04-28)

## Additions

- 2.2.2. Advance Download

## Changes

- 1.2. Glossary
    - Added entries for revised version and patch.
- 4.1.3. Using Extended Save Data
    - Revised the detailed `AccessibleSaveDataIds` specifications.
- 5.1.2. Creating Downloadable Applications
    - Added the `MANUAL_DISCLOSURE` build option.
- 5.2.3. Creating E-Manual CFA Files
    - Added the `–DMANUAL_DISCLOSURE` option.
- 6.1. Debugging Methods According to the Menu Displayed at Startup
    - Fixed the hierarchy of the Config tools menu to match the current tools.

# Version 1.1 (2015-01-15)

## Changes

- 4.1.3. Using Extended Save Data

- Fixed an error in `AccessibleSaveDataIds`.

- 4.1.4. Settings for Creating a Master ROM to Submit

  - Fixed the `MediaSize` item in the RSF file.

### Deletions

- Platform Notation

  - Deleted this page because the information about platform notation was moved to the Readme file.

## Version 1.0 (2014-09-04)

### Additions and Changes

- Initial version.

# 3DS Programming Manual: NFP Library

Version 1.4

**Nintendo Confidential**

This document contains confidential and proprietary information of Nintendo, and is protected under confidentiality agreements as well as the intellectual property laws of the United States and of other countries. No part of this document may be released, distributed, transmitted, or reproduced in any form , including by any electronic or mechanical means and by including within information storage and retrieval systems, without written permission from Nintendo.

©2015–2016 Nintendo. All rights reserved.

All company and product names in this document are the trademarks or registered trademarks of their respective companies.

CTR-06-0351-001-E

# 1. Introduction

This documentation provides developers of Nintendo applications that use the Nintendo Figurine Platform/NFC Featured Platform (NFP) with an overview of the NFP library, the functions used, and programming procedures.

Applications can use NFP to incorporate the use of branded character products called amiibo™ figures.

Note that this documentation omits the descriptions of NFP and amiibo figures. We recommend reading the **NFP Overview** and the **Planning and Operations Manual: NFP Edition** beforehand.

A separate error code list is provided with examples of the messages that are displayed when certain errors are returned by the NFP library. We recommend referring to this list as needed.

**Note:** CTR uses infrared to communicate with the NFC Reader/Writer.
To use the NFP library from a CTR application while another feature is using the infrared communication, you must end the other feature first.

Infrared communication is used by the following features.

- Infrared communication between systems
- Circle Pad Pro

CONFIDENTIAL

# 2. NFP Library Features

The NFP library is for reading and writing NFP tags (IC tags formatted for NFP) using the NFC (Near Field Communication) Reader/Writer for CTR or the built-in NFC module on the SNAKE system.

The NFP library currently supports the following features.

**Table 2-1. Features Supported by the NFP Library**

| | |
|---|---|

| Feature | Comments |
|---|---|
| NFC Tag Detection | The NFC Reader/Writer can detect amiibo, and SNAKE can detect amiibo and transport system IC cards (Suica, PASMO, Kitaca, TOICA, manaca, ICOCA, SUGOCA, nimoca, HAYAKAKEN).<br>This documentation refers to all detectable IC tags and cards as *NFC tags*.<br>NFC tags other than those listed above may also be detected, but there is no guarantee that the NFP library will operate. |
| Reading and Writing NFP Tags | The library can read from and write to NFP tags only. |
| Backing Up NFP Tags | When an application writes data to an NFP tag using the NFP library, the library automatically backs up the data to the system (under save data in the **amiibo Settings** application).<br>If the data on the NFP tag ever becomes corrupted, the application can explicitly restore the data on the NFP tag using the backup. |

No more than one tag can be used at a time.  Recognition of two or more tags at the same time is not supported and results in abnormal operation. Attempting to use the NFC Reader/Writer in this way may cause the NFC Reader/Writer to stop responding, and you will have to power-cycle the system.



# 3. Shared Region and Application-Specific Region

An NFP tag includes two data regions: a shared region and an application-specific region.

The shared region stores information about the amiibo itself, such as the character ID, and registration information. Any NFP-compatible application can read from the shared region, but only the **amiibo Settings** system application can write to the region.

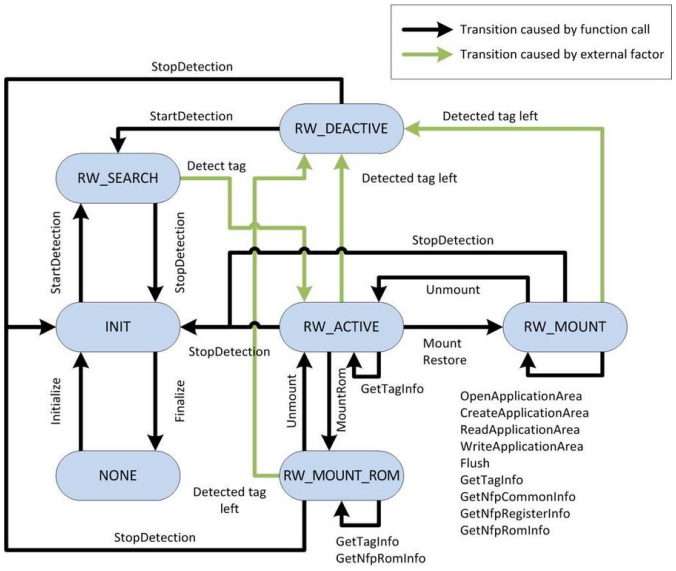
The application-specific region can hold up to 216 bytes of data. The application can decide what data to store. However, only one application can store data at a time. Data in the application-specific region can only be read by an application with the same access ID as the application that created the region. To overwrite the data in the application-specific region, prompt the user to delete the data using the **amiibo Settings** application and then create a new region.



# 4. State Transitions

The internal states of the NFP library transition as shown in the following diagram.

**Figure 4-1. NFP Library State Transition Diagram**



**Warning:** Although it is not indicated in the diagram, calling `nn::nfp::Finalize` in any state causes the library to transition to the `NONE` state. In addition, the state transitions to `INIT` whenever the system enters wireless-disabled mode or Sleep Mode in all cases except when the state is `NONE` or `INIT`.

**Warning:** When using the NFC Reader/Writer for CTR, if the connection to the device is lost while in a state other than `NONE` or `INIT`, the state transitions to `INIT`.

The states are defined in `nn::nfp::NfpState()` and you can get the current state by calling `nn::nfp::GetNfpState()`.

Each state represents the following.

**Table 4-1. State Definitions**

| State | Description |
|---|---|
| `NONE` | The state before the NFP library is initialized. |
| `INIT` | The NFP library is initialized but has not yet started looking for tags. |
| `RW_SEARCH` | The library is searching for tags. |
| `RW_ACTIVE` | A tag has been detected. |
| `RW_DEACTIVE` | The detected tag left the range of communication. |
| `RW_MOUNT` | The detected tag has been mounted. The detected tag cannot be mounted if it is not an NFP tag. |
| `RW_MOUNT_ROM` | The detected tag has been mounted. Unlike the `RW_MOUNT` state, this state limits the information that can be accessed. |

**Table 4-2. Functions Available in Each State**

| State | Available Functions (`nn::nfp` Namespace Omitted) |
|---|---|
| `NONE` | `Initialize` |
| `INIT` | `SetActivateEvent, SetDeactivateEvent, StartDetection` |
| `RW_SEARCH` | `StopDetection` |
| `RW_ACTIVE` | `GetTagInfo, Mount, MountRom, Restore, StopDetection` |
| `RW_DEACTIVE` | `StartDetection, StopDetection` |
| `RW_MOUNT` | `CreateApplicationArea, Flush, GetNfpCommonInfo, GetNfpRegisterInfo, GetNfpRomInfo, GetTagInfo, OpenApplicationArea, ReadApplicationArea, StopDetection, Unmount, WriteApplicationArea` |
| `RW_MOUNT_ROM` | `GetNfpRomInfo, GetTagInfo, StopDetection, Unmount` |
| Other than `NONE` | `Finalize, GetConnectionStatus, InitializeCreateInfo` |
| All | `GetNfpState` |

Using the NFP library increases the power consumption of the SNAKE system and affects the battery life. The relationship between the state of the NFP library and how it affects power consumption is as follows.

| State | SNAKE Battery Life |
|---|---|
| NONE | No change |
| INIT, RW_SEARCH, RW_DEACTIVE | Reduction of less than 1% |
| RW_ACTIVE, RW_MOUNT, RW_MOUNT_ROM | Reduction of approximately 8% |

Battery life information for the NFC Reader/Writer is as follows. The battery life for CTR is the same as when using infrared communication.

| State | Connection to NFC Reader/Writer | NFC Reader/Writer Battery Life |
|---|---|---|
| NONE, INIT | Disconnected | No change |
| INIT, RW_DEACTIVE | Connected | Reduction of approximately 40% |
| RW_SEARCH | Connected | Reduction of approximately 94% |
| RW_ACTIVE, RW_MOUNT, RW_MOUNT_ROM | Connected | Reduction of approximately 65 % |

We recommend transitioning to a state that minimizes power consumption whenever possible. However, it is unnecessary to call the `nn::nfp::Finalize()` function too frequently because the `nn::nfp::Initialize()` function requires approximately one second to execute.

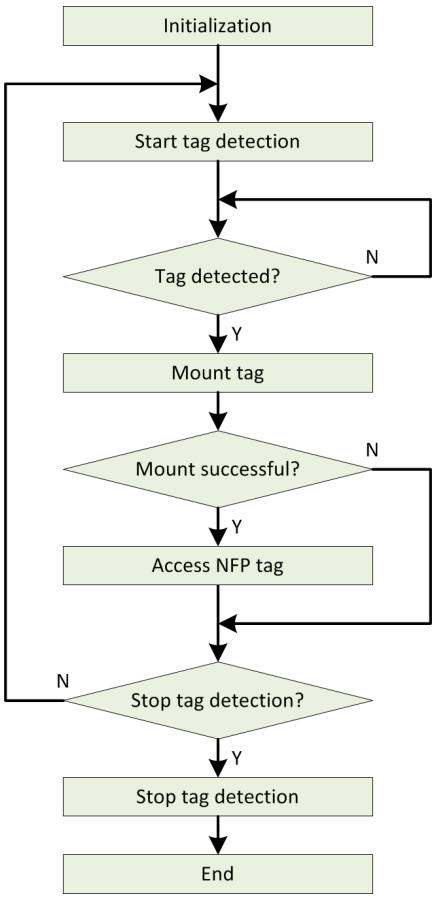


# 5. Process Flows

This chapter describes the process flow when using the NFP library. It includes an overview diagram and detailed sequences for each process.

The following diagram provides a simple overview of the NFP library process flow.

**Figure 5-1. Process Flow When Using the NFP Library (Overview)**

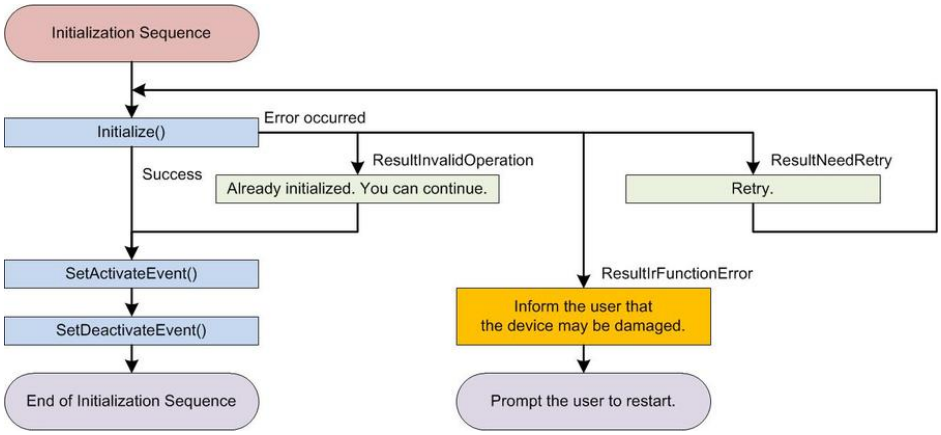



## 5.1. Initialization Sequence

This sequence is for initializing the NFP library.

Unless the NFP library is finalized while the application is still running, this sequence is only run once between starting the application and calling an NFP library function.

**Figure 5-2. Initialization Sequence Flowchart**

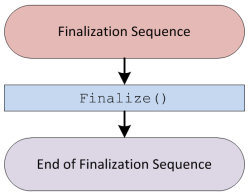



## 5.2. Finalization Sequence

This sequence is for finalizing the NFP library.

This sequence is run when the NFP library is no longer required by the application.

**Figure 5-3. Finalization Sequence Flowchart**



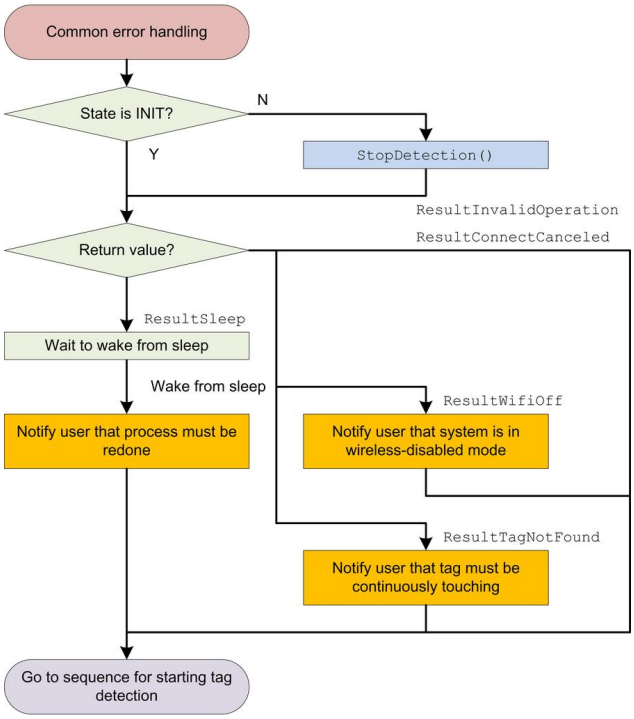

## 5.3. Common Error Handling

This is an example of the sequence used to handle errors that can be returned by any of the functions in the NFP library.

Errors handled with this sequence generally require retrying from the start of tag detection. This example assumes that the same tag will stay touching the device until communication finishes, and an error will occur when the tag is swapped with a different tag.

If the cause of the error can be addressed by the user, make sure to inform the user of how to fix the problem. For example, if the error is caused by the system being in wireless-disabled mode, prompt the user to enable wireless.

**Figure 5-4. Common Error Handling Flowchart**

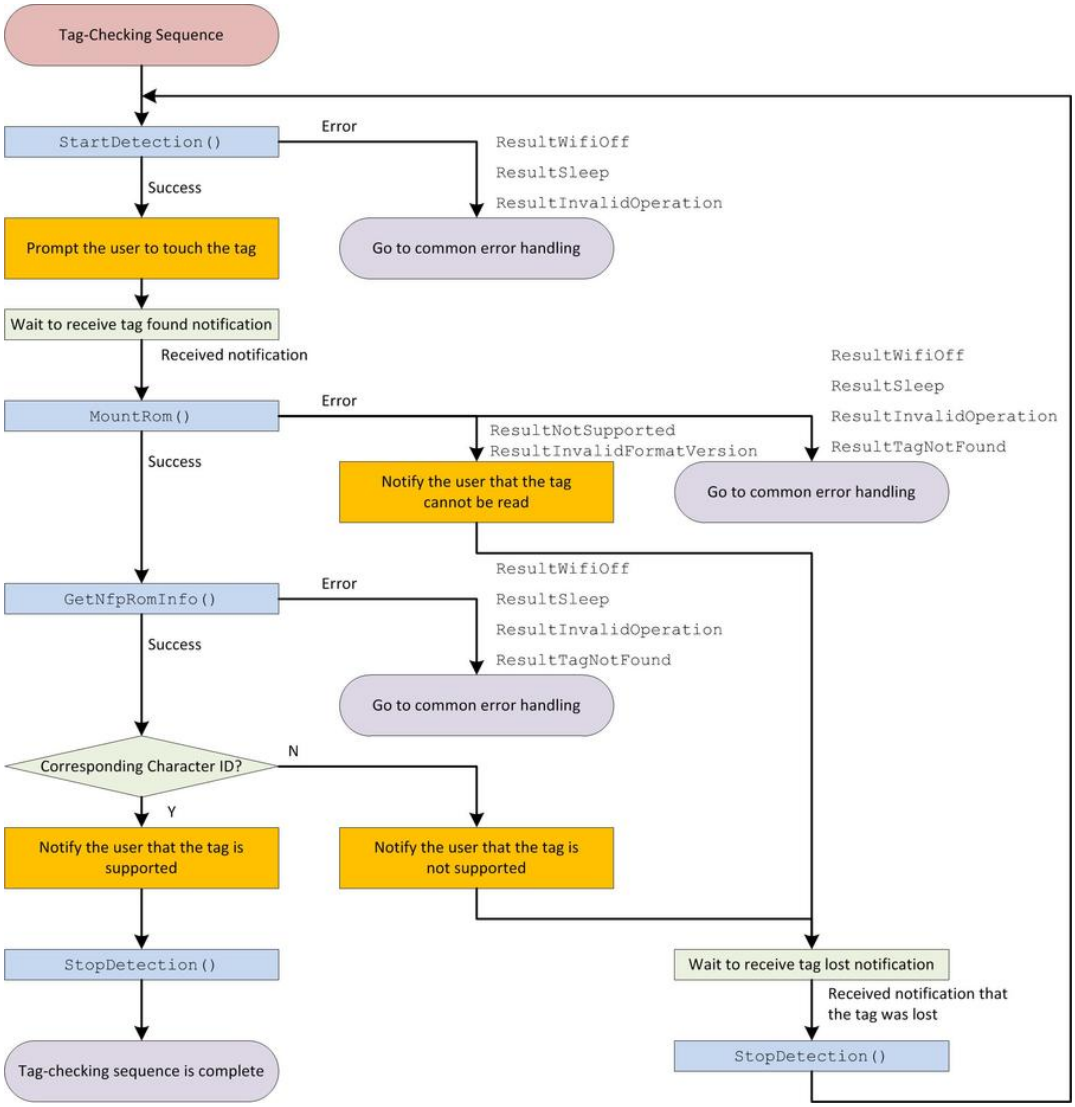



**Note:** If you are using the NFC Reader/Writer for CTR, proceed to the NFC Reader/Writer connection sequence instead of the tag detection start sequence.

## 5.4. Tag Check Sequence

This sequence checks which character the NFP tag belongs to in an application that does not access the owner information or the application-specific region.

**Figure 5-5. Tag Check Sequence Flowchart**

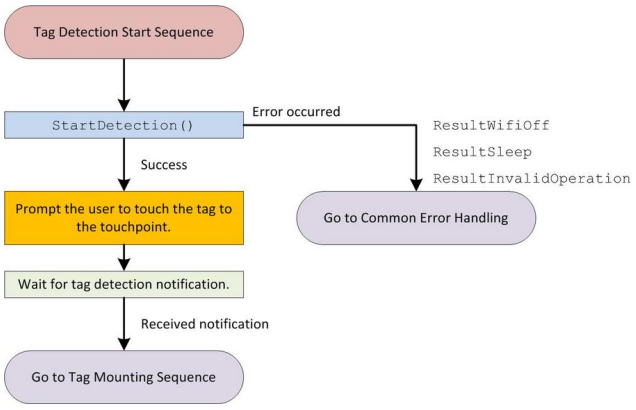



## 5.5. Tag Detection Start Sequence

This sequence sets the NFC module to start searching for tags and prompts the user to touch the tag to the touchpoint.

Although this step is not included in the flowchart, make sure to provide a way for the user to cancel when waiting for a tag detection notification.

**Figure 5-6. Tag Detection Start Sequence Flowchart**

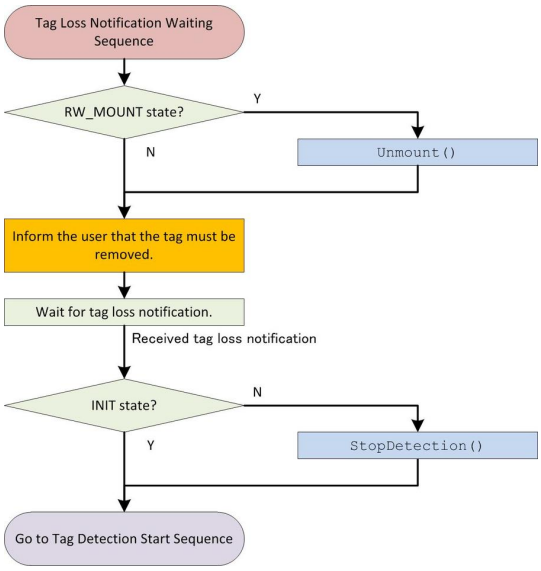


## 5.6. Tag Loss Notification Waiting Sequence

This sequence is followed when communication with the tag is complete, when a different tag is detected, and when the tag leaves the range of communication of the NFC module.

The flowchart assumes that the application will continue waiting for another tag, but the application does not need to wait for a tag loss notification if it does not intend to continue detecting tags after communication is complete.

**Figure 5-7. Tag Loss Notification Waiting Sequence Flowchart**



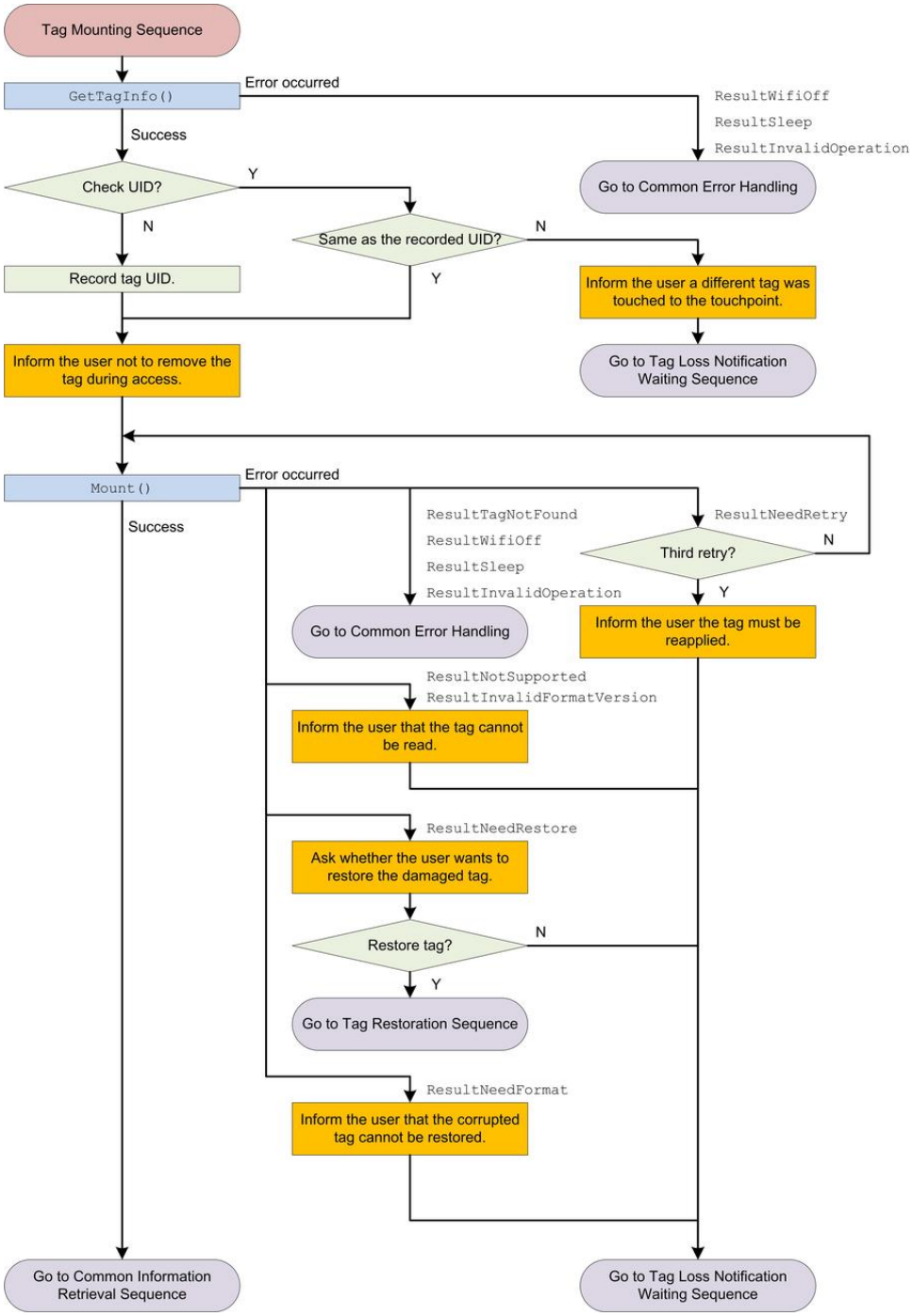

## 5.7. Tag Mounting Sequence

This sequence mounts a tag that was detected by the NFP library.

The flowchart retrieves the tag information and checks the tag UID before mounting the tag just in case any process needs to be resumed due to an error, or a tag that was read in a previous operation needs to be written to.

If a process error occurs and a retry is possible, you do not need to follow the number of attempts in the flowchart.

**Figure 5-8. Tag Mounting Sequence Flowchart**

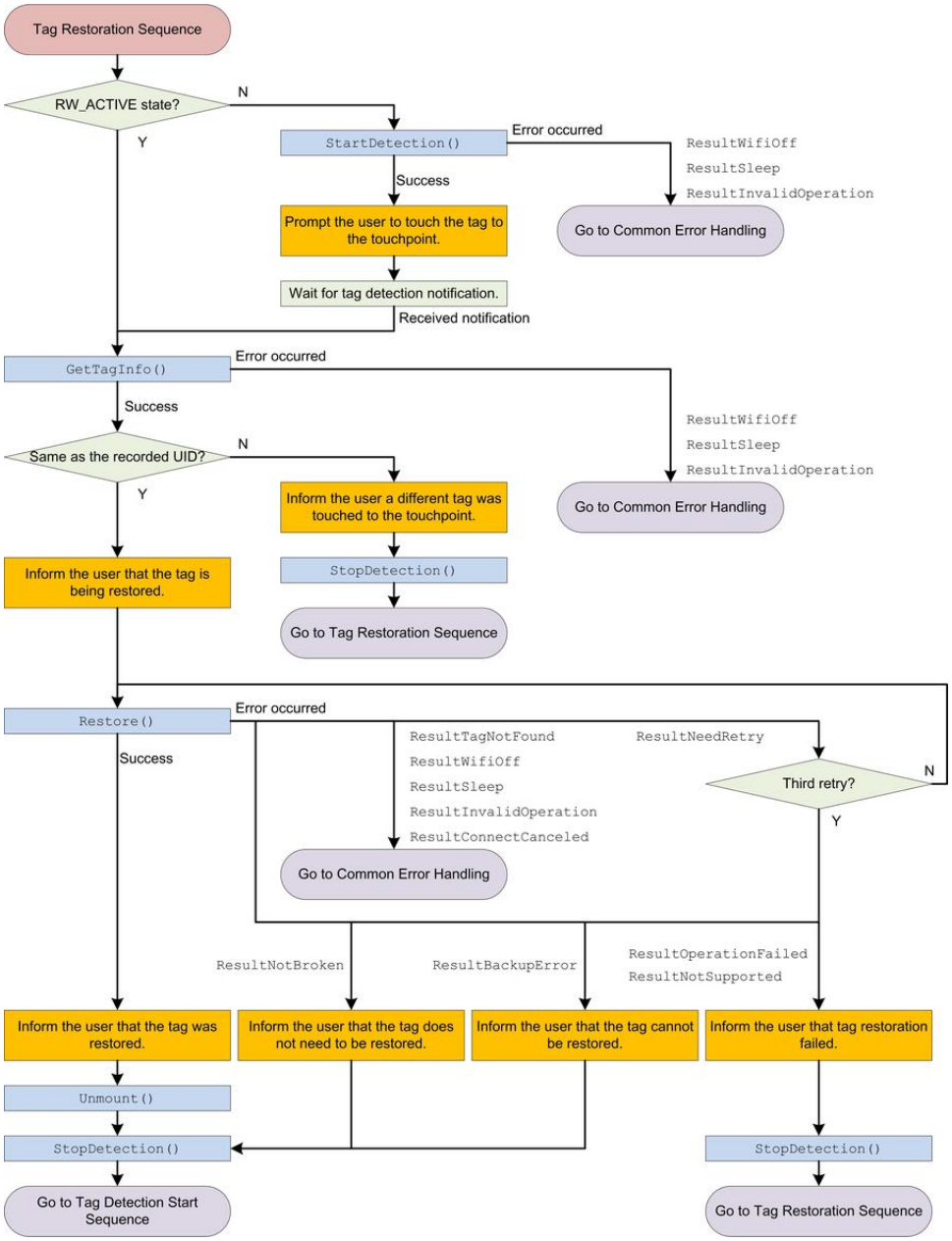



## 5.8. Tag Restoration Sequence

This sequence is used to restore a corrupted NFP tag that can still be restored in an application. You do not have to implement the following sequence to restore NFP tags. Instead, you can start **amiibo Settings** in recovery mode.

The flowchart is designed to restart from tag detection if mounting fails because the tag was removed or a different tag was touched to the touchpoint.

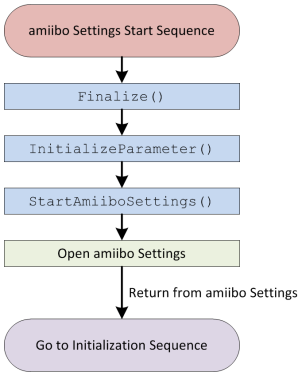
**Figure 5-9. Tag Restoration Sequence Flowchart**




## 5.9. amiibo Settings Start Sequence

This sequence starts **amiibo Settings**, which is used for initializing NFP tags, deleting the application-specific region, and other operations.

**Figure 5-10. amiibo Settings Start Sequence Flowchart**



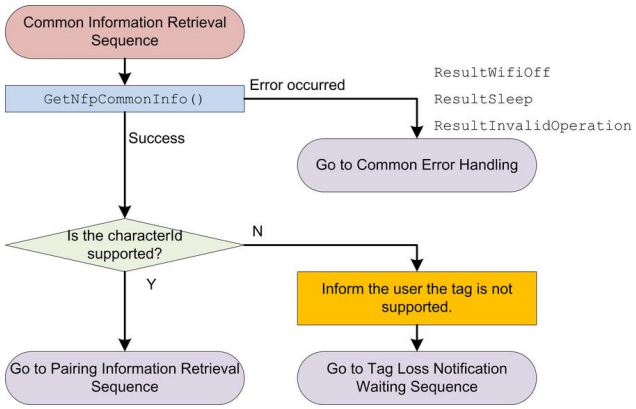

## 5.10. NFP Tag Access Process Flow

### 5.10.1. Common Information Retrieval Sequence

This sequence retrieves the common information recorded to the shared region of the NFP tag.

Determine whether the NFP tag is for a character that is supported by the application. If the character ID (`characterId`) is not supported by the application, inform the user that the tag is not supported.

**Figure 5-11. Common Information Retrieval Sequence Flowchart**




### 5.10.2. Registration Information Retrieval Sequence

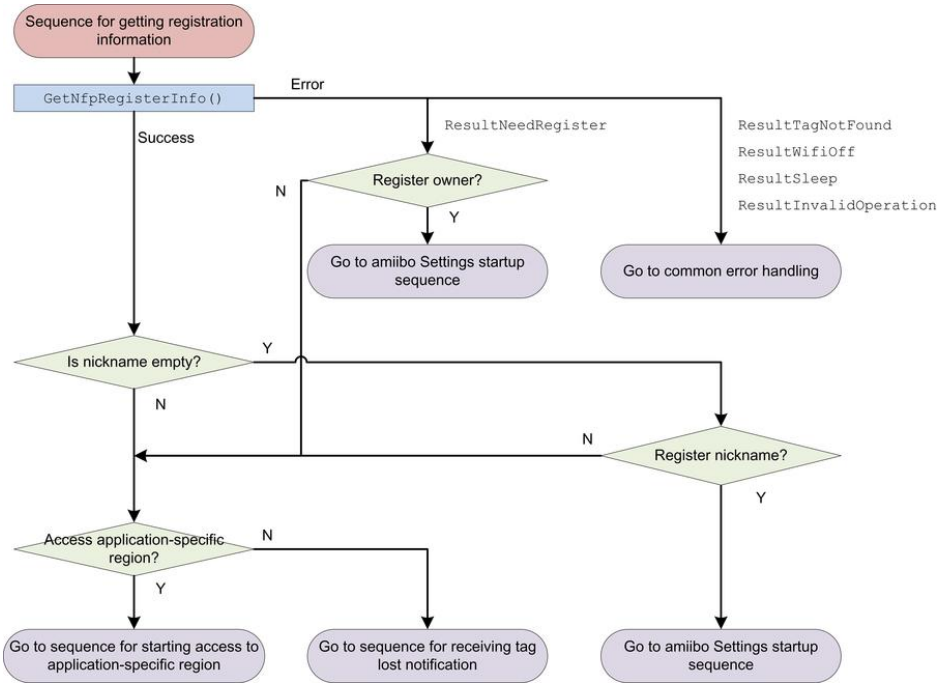
This sequence retrieves the registration information recorded to the shared region of the NFP tag.

The registration information includes the owner's Mii data and the amiibo nickname.

If your application requires the amiibo owner's Mii data and detects an NFP tag that does not have an owner registered, go to the sequence to start **amiibo Settings** so the user can register the owner information.

If your application requires an amiibo nickname and detects an NFP tag with an empty nickname, go to the sequence to start **amiibo Settings** so the user can register a nickname.

**Figure 5-12. Registration Information Retrieval Sequence Flowchart**




## 5.10.3. Application-Specific Region Access Start Sequence

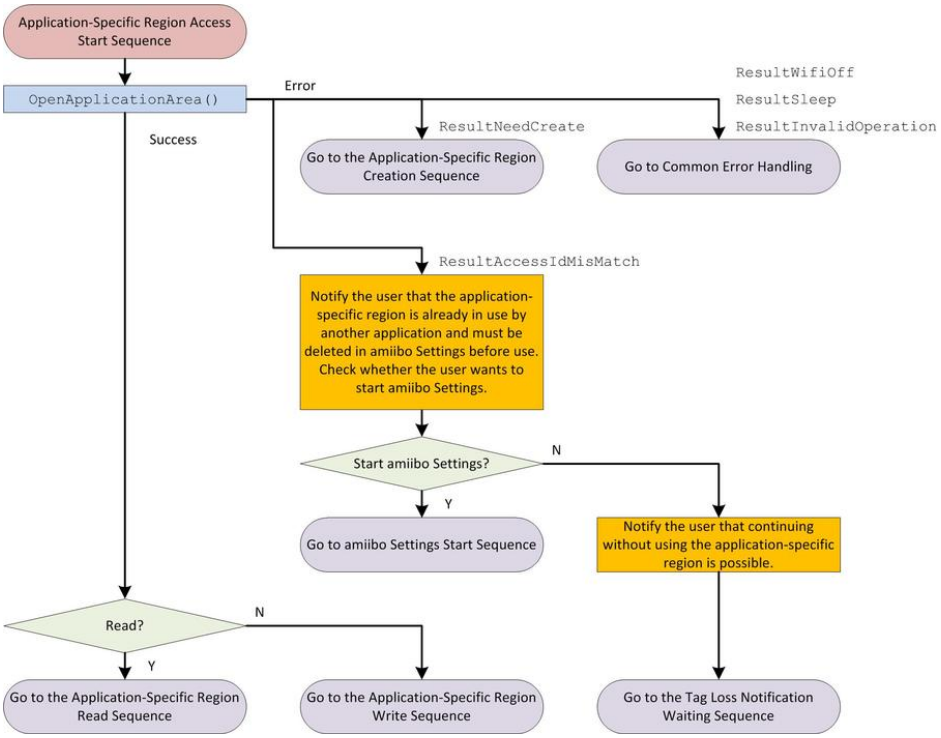
This sequence starts access to the application-specific region in the mounted NFP tag.

If the application-specific region has not yet been created in the NFP tag, go to the application-specific region creation sequence.

If the application-specific region has already been created by an application with a different access ID, the application cannot read from or write to the existing region. The existing region must first be deleted in **amiibo Settings**. Inform the user and start **amiibo Settings**.

We recommend informing the user of the differences in how the application works depending on whether it uses the application-specific region. In addition, implement your application in a way that can proceed even if the application-specific region is not used.

**Figure 5-13. Application-Specific Region Access Start Sequence Flowchart**


Application-Specific Region Access Start Sequence
OpenApplicationArea()
Error
ResultWifiOff
ResultSleep
ResultInvalidOperation
Success
ResultNeedCreate
Go to the Application-Specific Region Creation Sequence
Go to Common Error Handling
ResultAccessIdMisMatch
Notify the user that the application-specific region is already in use by another application and must be deleted in amiibo Settings before use. Check whether the user wants to start amiibo Settings.
Start amiibo Settings?
N
Y
Go to amiibo Settings Start Sequence
Notify the user that continuing without using the application-specific region is possible.
Read?
N
Y
Go to the Application-Specific Region Read Sequence
Go to the Application-Specific Region Write Sequence
Go to the Tag Loss Notification Waiting Sequence

## 5.10.4. Application-Specific Region Creation Sequence

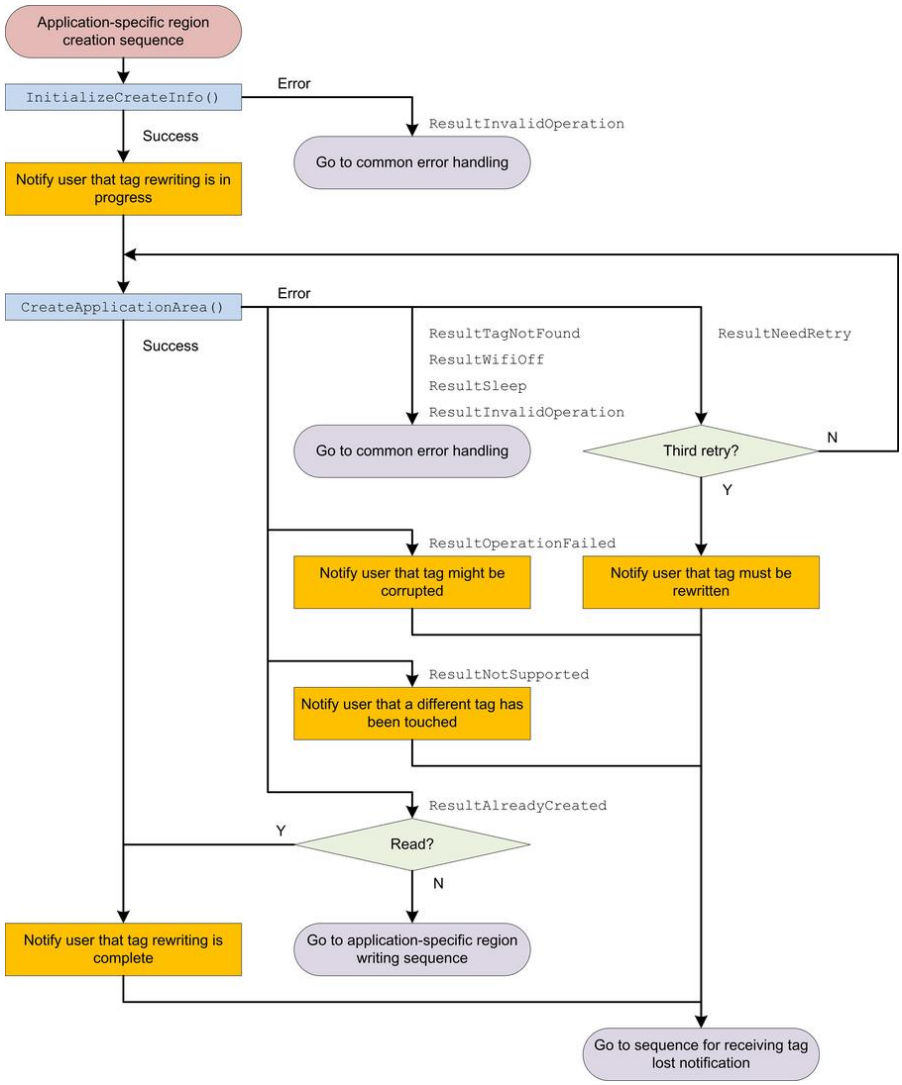
This sequence creates the application-specific region.

If this sequence was entered when the application was attempting to access the application-specific region to write data, you have the data written when `nn::nfp::InitializeCreateInfo` is called by passing the data to write as the initialization data to `nn::nfp::InitializeCreateInfo`.

If the `nn::nfp::CreateApplicationArea()` function returns that the application-specific region has already been created, the content of the region might not be what the application expected.

If a process error occurs and a retry is possible, you do not need to follow the number of attempts in the flowchart.

**Figure 5-14. Application-Specific Region Creation Sequence Flowchart**




## 5.10.5. Application-Specific Region Check Sequence

This sequence checks the integrity of the application-specific region immediately before writing to it.

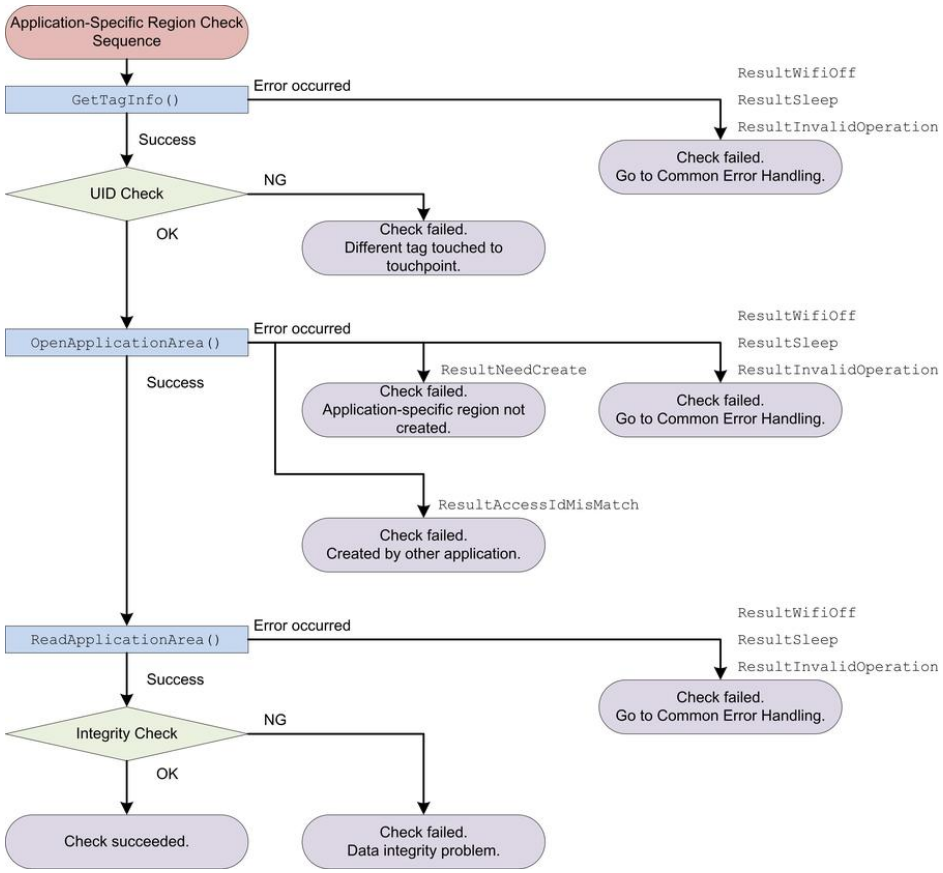
If there are periods of time when the application does not read from or write to the application-specific region, such as in an application that only reads and writes in between levels, the following kinds of actions can prevent the application from writing to the region. We recommend checking the integrity of the application-specific region before performing a write operation.

- A different NFP tag is touched to the touchpoint.
- The application-specific region is overwritten by another device.

- The application-specific region is deleted in **amiibo Settings**.

The following flowchart assumes the NFP tag has been successfully mounted.

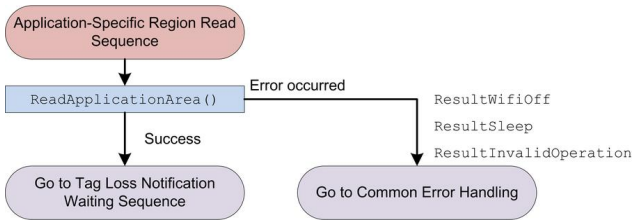
**Figure 5-15. Application-Specific Region Check Sequence Flowchart**




## 5.10.6. Application-Specific Region Read Sequence

This sequence is for reading from the application-specific region.

**Figure 5-16. Application-Specific Region Read Sequence Flowchart**



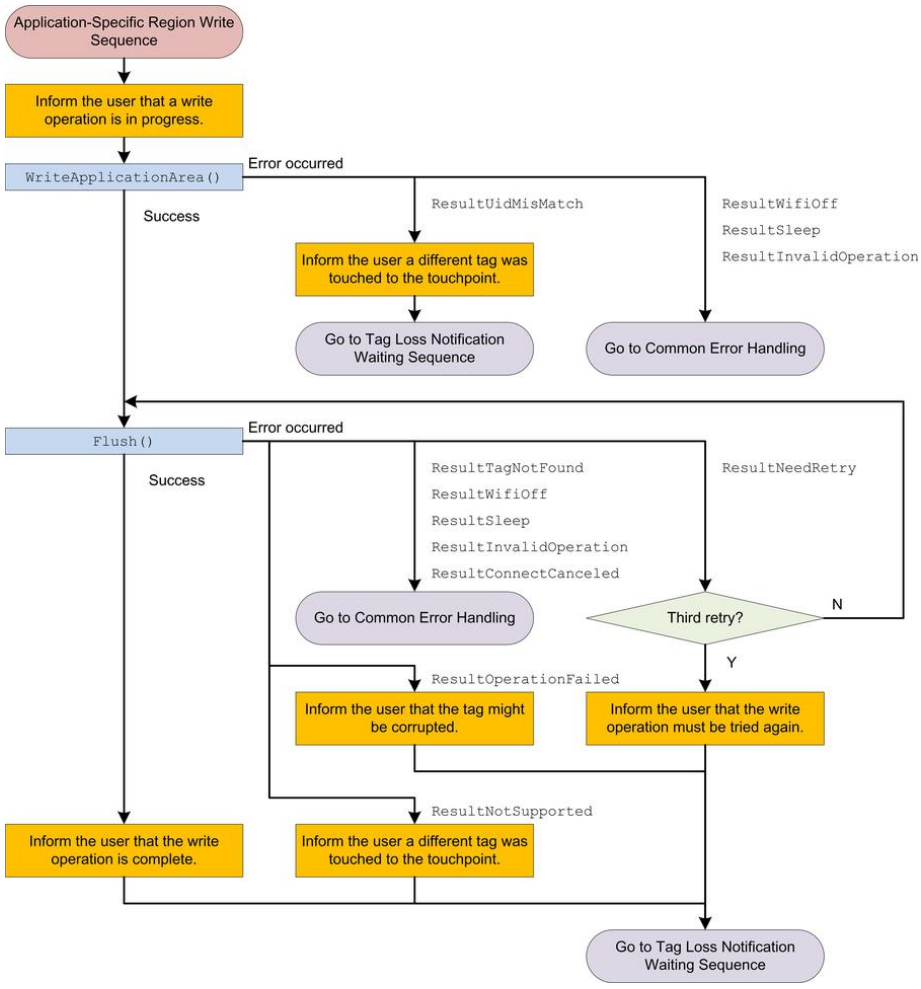

## 5.10.7. Application-Specific Region Write Sequence

This sequence is for writing data to the application-specific region.

In the flowchart, the application waits for the tag to leave the range of communication after the write process is complete, but if the application will not continue detecting tags, we recommend calling `nn::nfp::StopDetection` to stop tag detection until the application needs to start detecting tags again.

If a process error occurs and a retry is possible, you do not need to follow the number of attempts in the flowchart.

**Figure 5-17. Application-Specific Region Write Sequence Flowchart**




## 5.11. Required Processes in 3DS Applications

Applications on the 3DS require processes to handle various 3DS features, such as the HOME Button, Sleep Mode, and the POWER Button.

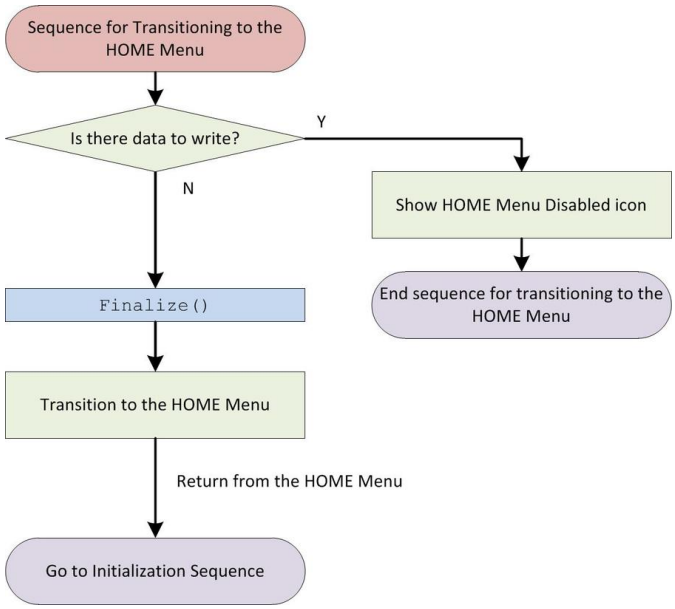
### 5.11.1. HOME Menu Transition Sequence

When the HOME Button is pressed, the application receives a request from the system to transition to the HOME Menu, and the application must determine whether to comply.

If the application cannot immediately transition to the HOME Menu because, for example, it has data that needs to be written, it can display the HOME Menu Disabled icon and reject the transition request.

You must finalize the NFP library to enable the use of NFC features by an applet started from the HOME Menu.

**Figure 5-18. HOME Menu Transition Sequence Flowchart**

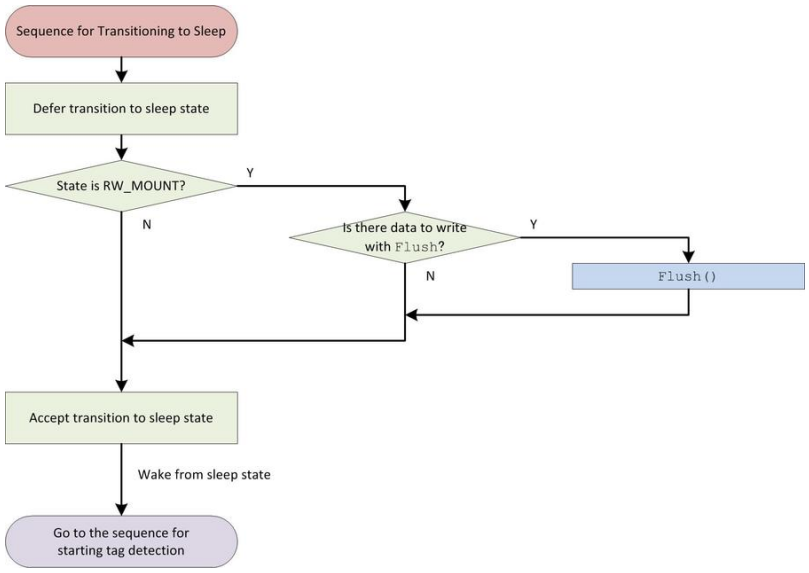

Sequence for Transitioning to the HOME Menu
Is there data to write?
Y
N
Show HOME Menu Disabled icon
End sequence for transitioning to the HOME Menu
Finalize()
Transition to the HOME Menu
Return from the HOME Menu
Go to Initialization Sequence

## 5.11.2. Sleep Mode Transition Sequence

When the system is closed (or when the Sleep switch is triggered on FTR), the application receives a request from the system to transition to Sleep Mode, and the application must determine whether to allow the transition.

If the application cannot immediately transition to Sleep Mode because, for example, it has data that needs to be written, it can hold the reply and continue processing until it is ready for the system to transition to Sleep Mode. When the application allows the transition to Sleep Mode, the system automatically stops tag detection and unmounts any tags. The application must then restart from the tag detection start sequence after waking from Sleep Mode.

**Figure 5-19. Sleep Mode Transition Sequence Flowchart**




## 5.11.3. Power Off Sequence

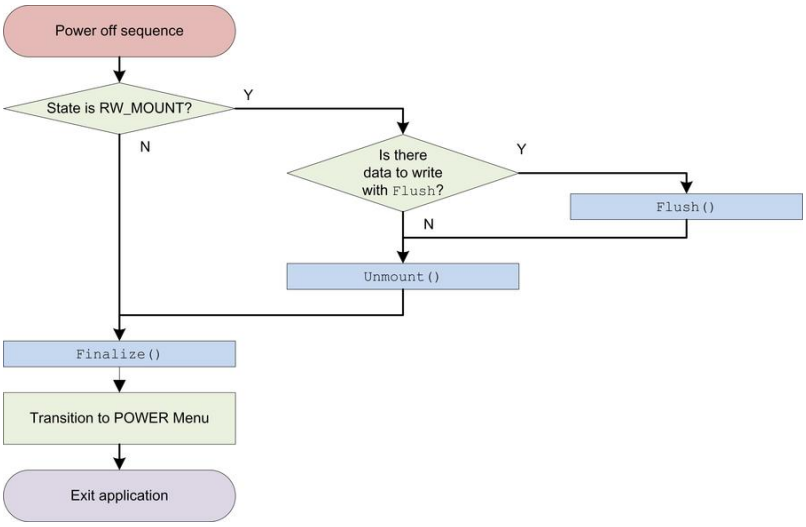
When the POWER Button is pressed, the application receives a request from the system to transition to the POWER Menu, and generally the application must immediately run the finalization process.

If some data still needs to be written, the application can continue the write process as long as it

can complete within the allotted finalization time.

The NFP library must be finalized before displaying the POWER Menu.

**Figure 5-20. Power Off Sequence Flowchart**




## 5.11.4. NFC Reader/Writer Connection Sequence

If you are using the NFC Reader/Writer for CTR, you must explicitly connect to the device from the application.

If the `nn::nfp::Connect()` function generates an error because the system has transitioned to Sleep Mode or Wireless-disabled mode, handle the error according to the value returned.
If you cannot connect to the NFC Reader/Writer, you must check the reason using the `nn::nfp::GetConnectResult()` function and handle the situation appropriately.

If a result of `nn::nfp::ResultUpdateRequired` is returned, you must start **amiibo Settings** to update the firmware of the NFC Reader/Writer.

If a result of `nn::nfp::ResultIrFunctionError` is returned, the infrared communication module on the CTR might be malfunctioning. When this error occurs, prompt the user to power cycle the CTR system, and to contact the Nintendo Customer Service Center if the problem persists.

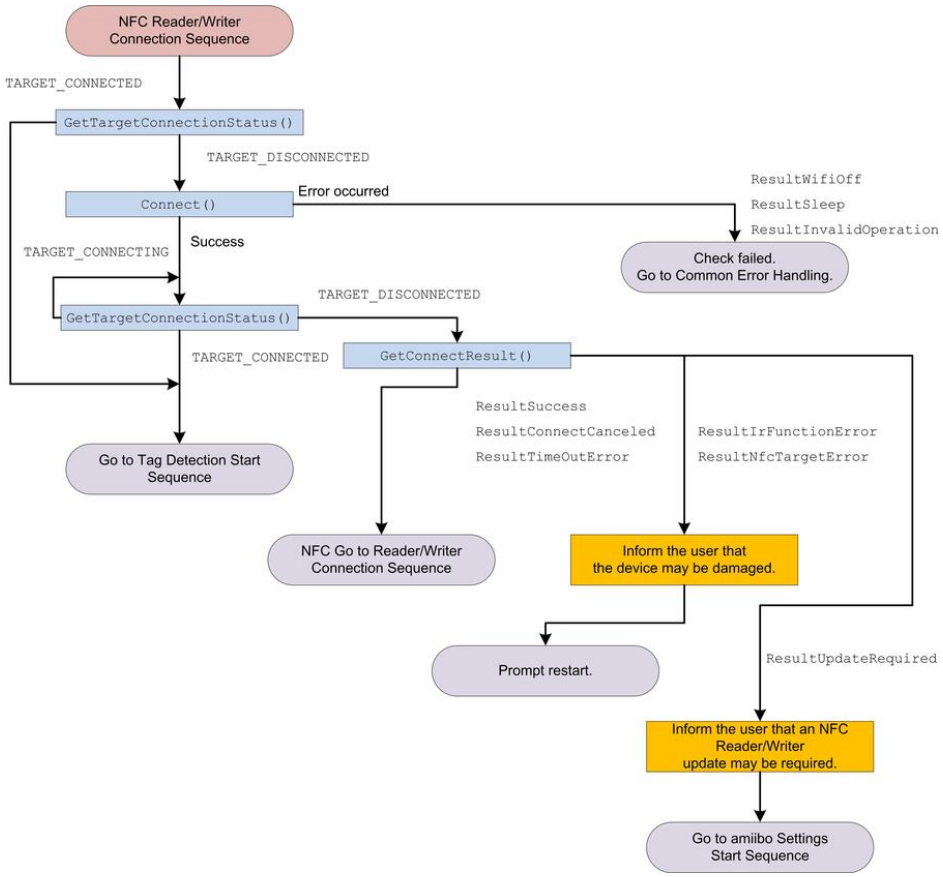
If a result of `nn::nfp::ResultNfcTargetError` is returned, the NFC Reader/Writer might be malfunctioning. When this error occurs, prompt the user to power cycle the NFC Reader/Writer and to contact the Nintendo Customer Service Center if the problem persists.

If a result of `nn::nfp::ResultConnectCanceled` is returned, either the connection was canceled or there was an internal error. Try connecting the NFC Reader/Writer again. Sometimes this error occurs because the application itself has called the `nn::nfp::Disconnect()` function and disconnected the device.

If a result of `nn::nfp::ResultTimeOutError` is returned, the NFC Reader/Writer did not respond. This can happen after you power cycle the NFC Reader/Writer or if it has been placed askew. Try connecting the NFC Reader/Writer again.

After a successful connection, if the NFC Reader/Writer can no longer communicate with the CTR because, for example, it was moved by the user, the state changes to disconnected after one second. Make sure that your application periodically calls `GetTargetConnectionStatus` to check whether the state has changed to `TARGET_DISCONNECTED`.

**Figure 5-21. NFC Reader/Writer Connection Sequence Flowchart**






# 6. Using the NFP Library

Note the following cautions when implementing an application that uses the NFP library.

- The NFP library may behave differently on SNAKE and CTR. Make sure that you check operation on both SNAKE and CTR.
- The NFP library operating mode on SNAKE uses only the on-board NFC. This is the same as when CRT-compatibility mode is forced by the debugger. Have CRT available when using the NFC Reader/Writer.
- The NFP library does not support multithreaded processes. Do not call NFP library functions simultaneously from multiple threads.
- A successful function call generally returns `nn::ResultSuccess`, but uses the `IsSuccess()` function or the `IsFailure()` function to determine whether execution succeeded.
- Return values belonging to `nn::nfp::ResultInvalidUsage` are implementation errors (`nn::nfp::ResultInvalidArgument`, `nn::nfp::ResultInvalidPointer`, and `nn::nfp::ResultBufferIsSmall`). Fix your implementation during development so that these errors are not returned.
- Many of the functions provided by the NFP library need a long time to finish. In particular, initializing the library can take upwards of five seconds and writing to an NFP tag can take one to two seconds. For this reason, we recommend calling the functions in the NFP library from a separate thread rather than from the main loop.
- After a tag is detected, if communication is lost because, for example, the tag is moved too far away from the reader, a loss-of-tag notification is sent to the application. At this point, the NFP library transitions to the `RW_DEACTIVE` state, and the mounted tag is unmounted.
  If you call an NFP library function in this state, most functions will return `nn::nfp::ResultInvalidOperation`. Be sure to have your application handle the tag loss notification and verify that the detected tag is still accessible before calling any NFP library functions.
- If the system enters Sleep Mode or wireless-disabled mode during the execution of an NFP library function, most functions will return `nn::nfp::ResultSleep` or `nn::nfp::ResultWifiOff`.

When these values are returned, the NFP library reverts to the INIT state, tag detection stops, and the mounted tag is unmounted. This state is the same as the one that is entered when the nn::nfp::Unmount() and nn::nfp::StopDetection() functions are called.
Even if the system wakes from Sleep Mode or transitions back to wireless-enabled mode, the NFP library does not automatically return to its previous state. The application must resume tag detection by calling nn::nfp::StartDetection again.

- Before transitioning to the HOME Menu or Sleep Mode, if the nn::nfp::Flush() function has not yet been called when writing to the application-specific region, you can delay the transition to the HOME Menu or Sleep Mode until the write operation has been completed by calling nn::nfp::Flush. An alternative approach is to display the HOME Menu Disabled icon and prevent transitioning to the HOME Menu while writing to the NFP tag.
- Before transitioning to the HOME Menu or an applet, call nn::nfp::Finalize to put the NFP library into the NONE state.

The NFP library includes functions that access backup data and NFP tags. Calling these functions excessively can shorten the life of the hardware and NFP tag.

Implement your application so that it does not call the following functions too often, unless the user intentionally repeats an operation that requires them.

**Table 6-1. Functions That Access Backup Data or NFP Tags**

| Function | Read From Backup Data | Write to Backup Data | Write to NFP Tag |
|---|---|---|---|
| nn::nfp::Initialize | Always | When there is no save data for backups | None |
| nn::nfp::Mount | Always | When mounting a tag for the first time, or when mounting a tag that was overwritten by another device | None |
| nn::nfp::Restore | Always | None | Always |
| nn::nfp::Flush | None | Always | Always |
| nn::nfp::CreateApplicationArea | None | Always | Always |

## 6.1. Initialization

Call the nn::nfp::Initialize() function to initialize the NFP library. Nintendo strongly recommends calling this function outside of the main thread because it usually takes about several hundred milliseconds to execute, and about five seconds when run for the first time on the system. We also recommend calling this function after the application is ready to update the screen instead of during startup.

Almost all of the functions in the NFP library return nn::nfp::ResultInvalidOperation if called before the library is initialized. If you call nn::nfp::Initialize when the library is already initialized, it returns nn::nfp::ResultInvalidOperation, but you can handle this result as a success in this case.

If initialization of the NFP library fails, sometimes nn::nfp::ResultNeedRetry is returned. When this occurs, call nn::nfp::Initialize again. You can increase the chance of success on the first try if you wait around one second before calling.

**Table 6-2. Possible Return Values From nn::nfp::Initialize**

| Return Value | Description |
|---|---|
| nn::ResultSuccess | Initialization successful. |
| nn::nfp::ResultInvalidOperation | Already initialized. |
| nn::nfp::ResultNeedRetry | Try again. |
| nn::nfp::ResultIrFunctionError | The IrDA module may have been damaged. |

### 6.1.1. Setting Events to Receive Tag Detection Notifications

Before starting tag detection, you must set events so the application can receive tag detection notifications from the NFP library.

The NFP library sends tag detection notifications to the application at the following times.

- A tag was detected within the range of communication of the NFC module (tag detection notification).
- The detected tag left the range of communication (tag loss notification).
- The `nn::nfp::StopDetection`, `nn::nfp::Disconnect` or `nn::nfp::Finalize()` function was called when a tag had been detected (tag loss notification).

Use the `nn::nfp::SetActivateEvent()` function to set the event for tag detection notifications and the `nn::nfp::SetDeactivateEvent()` function for tag loss notifications.

Each of these functions takes a pointer to an instance of the `nn::os::Event` class as an argument. The NFP library manages the events, so the application does not need to initialize or finalize the instances that are passed.

These functions must be called before starting tag detection (while the state is `INIT`).

**Table 6-3. Possible Return Values From nn::nfp::SetActivateEvent and nn::nfp::SetDeactivateEvent**

| Return Value | Description |
|---|---|
| `nn::ResultSuccess` | The event was successfully set. |
| `nn::nfp::ResultInvalidPointer` | The pointer passed as an argument is invalid. |
| `nn::nfp::ResultInvalidOperation` | The library is not in an appropriate state for calling this function. |

### 6.1.1.1. Handling Tag Detection Notifications

When the NFP library detects a tag, the application is notified by putting the event object set with `nn::nfp::SetActivateEvent` into a signaled state.

When a tag is detected, the following processes become available.

- Getting Tag Information
- Mounting a Tag

### 6.1.1.2. Handling Tag Loss Notifications

When the NFP library loses a tag because the tag leaves the range of communication, the application is notified by putting the event object set with `nn::nfp::SetDeactivateEvent` into a signaled state. The NFP library transitions to the `RW_DEACTIVE` state as soon as the notification is received. To continue tag detection, you must call the `nn::nfp::StartDetection()` function again.

You can determine the cause of the tag loss based on the tag connection status that can be retrieved using `nn::nfp::GetConnectionStatus`. The tag connection status is defined with the `nn::nfp::ConnectionStatus` structure. A pointer to the structure must be passed to `nn::nfp::GetConnectionStatus`.

If `nn::nfp::GetConnectionStatus` succeeds, the connection status of the current tag will be stored in the structure that was passed as an argument. The detailed connection status (reason for disconnect) is defined by the `nn::nfp::DeactivateReason` enumerator and stored in the `deactivateReason` member of the structure.

**Table 6-4. Detailed Connection Status (Reason for Disconnect)**

| Definition | Connection Status | Description |
|---|---|---|
| `BEFORE_ACTIVATION` | Not found | A tag has not been detected yet. |
| `TAG_IS_ACTIVE` | Connected | The tag can be accessed. |
| `DETECT_ERROR` | Connection failure | Not recognized as a valid tag. The tag data may be corrupted. |
| `CONNECTION_LOST` | Disconnected | Could not maintain the connection because the tag left the range of communication or the signal |

| | | |
|---|---|---|
| | | is too weak. |
| UNKNOWN_DEACTIVATE_REASON | Disconnected | The connection was lost for a reason not recognized by the library. |

**Table 6-5. Possible Return Values From nn::nfp::GetConnectionStatus**

| Return Value | Description |
|---|---|
| nn::ResultSuccess | The connection status was successfully retrieved. |
| nn::nfp::ResultInvalidPointer | The process failed due to a problem in the argument that was specified. This result is an implementation error that must be fixed in the development phase. |
| nn::nfp::ResultInvalidOperation | The library is not in an appropriate state for calling this function. |

## 6.2. Starting Tag Detection

Call the nn::nfp::StartDetection() function to start tag detection.

While tag detection is running, the NFP library periodically checks whether any tags have been detected and whether it can communicate with any detected tags. There is a time lag (up to 500 milliseconds on the SNAKE system or up to about one second on the CTR system) between the occurrence of the event that causes the tag detection state to change and when the application is notified of the tag detection. There also is a time lag of up to about two seconds until the application is notified of a tag loss. Do not implement your application in a way that requires careful timing when applying and removing tags. This assumes that the tag is placed in the center of the lower screen on the SNAKE system. Specifically, the tag can be accurately recognized within ±10 mm in the horizontal and vertical directions from the center of the LCD screen, and within 5 mm of the surface. This is the same with CLOSER. Avoid gameplay and user interface designs that encourage the user to place a tag away from the center of the LCD screen.

Also note that the electromagnetic waves used for tag detection on the SNAKE system cause the input from the touch panel to become unstable. Use of the touch panel for input during detection is restricted on SNAKE due to the potential for unintended operations in an application. Specifically, the status member of nn::hid::CTR::TouchPanelStatus is always 0 (pen up) from the time the nn::nfp::StartDetection() function is called until the nn::nfp::StopDetection() or nn::nfp::Finalize() function is called to stop tag detection.

There is no such restriction on the CTR because the NFC Reader/Writer is used for tag detection. However, if your application uses the touch panel on the CTR during tag detection, make sure that the application can also be controlled on the SNAKE by providing a different UI or other method.

In the NFP library, the power consumption and heat generated is greatest at the moment that the tag is detected. If nn::nfp::StartDetection() and nn::nfp::StopDetection() are repeated when the tag is placed, tag detection will occur repeatedly over a short period. This can cause SNAKE to generate more heat than it is designed to handle and may destabilize the system. This problem does not occur on CTR, but the power consumption is still high. For this reason, restrictions have been put in place for both SNAKE and CTR to prevent high-frequency execution of nn::nfp::StartDetection(). Specifically, nn::nfp::StartDetection() can be executed 10 times at any point during a 10-second period, but it cannot be executed 11 or more times during that period. This has been implemented by blocking the execution of nn::nfp::StartDetection() until 10 seconds have passed since the 10th previous execution of nn::nfp::StartDetection(). Execution is not blocked if 10 seconds have already passed, or if the function has not yet been executed 10 times.

**Table 6-6. Possible Return Values From nn::nfp::StartDetection**

| Return Value | Description |
|---|---|
| nn::ResultSuccess | Tag detection has started. |
| nn::nfp::ResultInvalidOperation | The library is not in an appropriate state for calling this function. |
| nn::nfp::ResultSleep | The process failed because the system is entering Sleep Mode. |
| nn::nfp::ResultWifiOff | The process failed because the system is in wireless-disabled mode. |

## 6.3. Getting Tag Information

Call the `nn::nfp::GetTagInfo()` function to get tag information. Tag information is defined with the `nn::nfp::TagInfo` structure. A pointer to the structure must be passed to `nn::nfp::GetTagInfo`.

**Table 6-7. Possible Return Values From nn::nfp::GetTagInfo**

| Return Value | Description |
|---|---|
| `nn::ResultSuccess` | The tag information was successfully retrieved. |
| `nn::nfp::ResultInvalidPointer` | The process failed due to a problem in the argument that was specified. This result is an implementation error that must be fixed in the development phase. |
| `nn::nfp::ResultInvalidOperation` | The library is not in an appropriate state for calling this function. |
| `nn::nfp::ResultSleep` | The process failed because the system is entering Sleep Mode. |
| `nn::nfp::ResultWifiOff` | The process failed because the system is in wireless-disabled mode. |

**Note:** The NFP library can only get the tag UID from the tag information. We recommend using `nn::nfp::GetTagInfo` to get the tag information in advance for applications that access the application-specific region of the NFP tag.

Because the tag information only contains information that can be retrieved from any tag type that is accessible to the NFP library, there is no way to determine whether the detected tag is an NFP tag.

## 6.4. Mounting a Tag

To access the information in an NFP tag, you must first call `nn::nfp::Mount` or `nn::nfp::MountRom` to mount the detected tag.

Neither function will mount a tag that is not an NFP tag, but the `nn::nfp::MountRom()` function can mount a tag even if it has corrupted data, although there are limitations on the information that can be accessed.

You can use the return value to determine whether a tag is an NFP tag, as described below.

**Table 6-8. Tag Statuses and Function Return Values**

| Tag Status | `nn::nfp::Mount` | `nn::nfp::MountRom` |
|---|---|---|
| Not an NFP tag | `nn::nfp::ResultNotSupported` | |
| NFP tag with unknown format version | `nn::nfp::ResultInvalidFormatVersion` | |
| NFP tag with normal data | `nn::ResultSuccess` | |
| NFP tag with corrupted data (No backup data for the tag) | `nn::nfp::ResultNeedFormat` | `nn::ResultSuccess` |
| NFP tag with corrupted data (Backup data for the tag is available) | `nn::nfp::ResultNeedRestore` | `nn::ResultSuccess` |

If the detected tag is not an NFP tag, a value of `nn::nfp::ResultNotSupported` is returned.

If the detected tag is an NFP tag but cannot be mounted because the format version is unknown, a value of `nn::nfp::ReultInvalidFormatVersion` is returned.

If the detected tag is an NFP tag and is successfully mounted, a value of `nn::nfp::ResultSuccess` is returned.

If the detected tag is an NFP tag and the data is corrupted, a value of `nn::nfp::ResultNeedRestore` or `nn::nfp::ResultNeedFormat` is returned. If the first value is returned, you can restore the tag data in the application by calling `nn::nfp::Restore`. If the second value is returned, you cannot restore the tag data within the application, so the NFP tag must be reset in **amiibo Settings**.

When `nn::nfp::ResultTagNotFound` is returned, the NFP library transitions to the `RW_DEACTIVE`

state as soon as the notification is received. To continue tag detection, you must call the `nn::nfp::StartDetection()` function again.

When `nn::nfp::ResultNeedRetry` is returned, the mounting process might succeed if `nn::nfp::Mount` is called again.

> **Note:** Up to about 500 milliseconds is needed to mount a tag.
>
> If you want to try again, we recommend a retry interval of around 100 milliseconds.

**Table 6-9. Possible Return Values From nn::nfp::Mount**

| Return Value | Description |
|---|---|
| `nn::ResultSuccess` | The tag was successfully mounted. |
| `nn::nfp::ResultInvalidOperation` | The library is not in an appropriate state for calling this function. |
| `nn::nfp::ResultSleep` | The process failed because the system is entering Sleep Mode. |
| `nn::nfp::ResultWifiOff` | The process failed because the system is in wireless-disabled mode. |
| `nn::nfp::ResultNotSupported` | The tag is not an NFP tag. |
| `nn::nfp::ResultNeedRestore` | The tag data is corrupted. Call `nn::nfp::Restore` to restore the tag data. |
| `nn::nfp::ResultNeedFormat` | The tag data is corrupted and must be reset in **amiibo Settings**. |
| `nn::nfp::ResultTagNotFound` | The tag could not be found. This result is caused by the tag leaving the range of communication or being swapped with a different tag. |
| `nn::nfp::ResultNeedRetry` | An error occurred during mounting. The process might succeed if tried again. |
| `nn::nfp::ResultInvalidFormatVersion` | The tag version is not supported. Inform the user that the tag is not supported. |

**Table 6-10. Possible Return Values From nn::nfp::MountRom**

| Return Value | Description |
|---|---|
| `nn::ResultSuccess` | The tag was successfully mounted. |
| `nn::nfp::ResultInvalidOperation` | The library is not in an appropriate state for calling this function. |
| `nn::nfp::ResultSleep` | The process failed because the system is entering Sleep Mode. |
| `nn::nfp::ResultWifiOff` | The process failed because the system is in wireless-disabled mode. |
| `nn::nfp::ResultNotSupported` | The tag is not an NFP tag. |
| `nn::nfp::ResultTagNotFound` | The tag could not be found. This result is caused by the tag leaving the range of communication or being swapped with a different tag. |
| `nn::nfp::ResultInvalidFormatVersion` | The tag version is not supported. Inform the user that the tag is not supported. |

### 6.4.1. Restoring Tags

When the `nn::nfp::Mount()` function returns `nn::nfp::ResultNeedRestore`, you can restore the NFP tag from the application by calling the `nn::nfp::Restore()` function.

If a value of `nn::nfp::ResultNeedRetry` is returned, the tag restoration process might succeed if `nn::nfp::Restore` is called again.

> **Note:** Restoring a tag can take longer than a second.
>
> If you want to try again, we recommend a retry interval of around 100 milliseconds.

The tag is restored using the data that was backed up by the NFP library when mounting the tag or immediately before writing to the tag. Consequently, the restored data is the data that the

application is attempting to write, and not the data that was in the tag before the write process failed.

**Table 6-11. Possible Return Values From nn::nfp::Restore**

| Return Value | Description |
|---|---|
| `nn::ResultSuccess` | The tag was successfully restored. |
| `nn::nfp::ResultInvalidOperation` | The library is not in an appropriate state for calling this function. |
| `nn::nfp::ResultSleep` | The process failed because the system is entering Sleep Mode. |
| `nn::nfp::ResultWifiOff` | The process failed because the system is in wireless-disabled mode. |
| `nn::nfp::ResultNotSupported` | The tag is not an NFP tag. |
| `nn::nfp::ResultNotBroken` | Restoration is unnecessary because the tag is not corrupted. |
| `nn::nfp::ResultTagNotFound` | The tag could not be found. This result is caused by the tag leaving the range of communication or being swapped with a different tag. |
| `nn::nfp::ResultOperationFailed` | An error occurred during writing. The tag data might not have been restored correctly. |
| `nn::nfp::ResultNeedRetry` | An error occurred during the write process. The function must be called again. |
| `nn::nfp::ResultConnectCanceled` | The NFC Reader/Writer was disconnected due to an internal error. |
| `nn::nfp::ResultBackupError` | Failed to access the backup data. This NFP tag cannot be restored. |

## 6.5. Starting amiibo Settings

The following steps are required when starting **amiibo Settings**.

1. Configure the information required to start **amiibo Settings**.
2. Finalize the NFP library.
3. Start **amiibo Settings.**

Start by configuring the `nn::nfp::Parameter` structure with the information required to start **amiibo Settings**.

Call `nn::nfp::InitializeParameter` to initialize the structure and then configure the `input` member (`nn::nfp::Input` structure) with the information required when starting **amiibo Settings** from an application.

When **amiibo Settings** detects the tag, it must determine whether it is the same tag specified by the application. Be sure to specify the tag information previously retrieved using `nn::nfp::GetTagInfo` in the `tagInfo` member.

Configure the other members (`mode`, `isRegistered`, `registerInfo`, and `commonInfo`) with the following values based on the use case.

**Table 6-12. Values to Use When Starting amiibo Settings by Use Case**

| Use Case | mode | isRegistered | registerInfo |
|---|---|---|---|
| The `nn::nfp::Mount()` function returned `nn::nfp::ResultNeedRestore`. | `AMIIBO_SETTINGS_RESTORE` | `false` | `NULL` |
| The `nn::nfp::GetNfpRegisterInfo()` function returned `nn::nfp::ResultNeedRegister`. | `AMIIBO_SETTINGS_NICKNAME_OWNER` | `false` | `NULL` |
| The `nn::nfp::GetNfpRegisterInfo()` function returned `nn::nfp::ResultSuccess`, but the nickname was an empty string. | `AMIIBO_SETTINGS_NICKNAME_OWNER` | `true` | Information retrieved using `nn::nfp::GetNfpRegisterInfo` |
| The `nn::nfp::OpenApplicationArea()` function returned `nn::nfp::ResultAccessIdMisMatch`, | `AMIIBO_SETTINGS_ERASE_GAMEDATA` | `true` | Information retrieved using `nn::nfp::GetNfpRegisterInfo` |

| | | | |
|---|---|---|---|
| so you want to delete the application-specific region. | | | |

The mode member takes an enumerator of the nn::nfp::AmiiboSettingsMode enumeration and specifies the mode to use when starting **amiibo Settings**.

**Table 6-13. amiibo Settings Startup Modes**

| Definition | Description |
|---|---|
| AMIIBO_SETTINGS_RESTORE | Restores a tag. |
| AMIIBO_SETTINGS_NICKNAME_OWNER | Registers the owner and sets a nickname. |
| AMIIBO_SETTINGS_ERASE_GAMEDATA | Erases the application-specific region. |

Call the nn::nfp::StartAmiiboSettings() function to start **amiibo Settings**. Before calling this function, you must finalize the NFP library by calling nn::nfp::Finalize. Consequently, the NFP library must be reinitialized before use after returning from **amiibo Settings**.

If the nn::nfp::StartAmiiboSettings returns false, the startup of **amiibo Settings** failed. This result could be caused by the following factors.

- The NFP library was not finalized (by calling nn::nfp::Finalize).
- The **amiibo Settings** applet is not installed (nn::nfp::IsAmiiboSettingsAvailable returns false).
- The parameter settings are incorrect.
- The applet failed to start.

The Parameter.output.result object stores whether any changes were made to the NFP tag in **amiibo Settings**. A value of nn::nfp::AMIIBO_SETTINGS_RESULT_SUCCESS in this member indicates that changes were made to the NFP tag in **amiibo Settings**, while any other value indicates that no changes were made because the operation was canceled by the user or other reason.

Similar to other library applets, when control returns from nn::nfp::StartAmiiboSettings(), use the nn::applet::IsExpectedToProcessPowerButton() and nn::applet::IsExpectedToCloseApplication() functions to check whether the application has been requested to close because the POWER Button was pressed while the applet was running.

## 6.6. Accessing the NFP Tag

When a tag is successfully mounted, you can access the following information in the NFP tag.

- Shared region
- Application-specific region

> **Warning:** The application cannot access the application-specific region unless it has the same access ID as the application that created the region.
>
> If the tag was mounted with nn::nfp::MountRom, the application can only access part of the shared region.

### 6.6.1. Accessing the Shared Region

Before using the information in the mounted NFP tag, use nn::nfp::GetNfpCommonInfo or nn::nfp::GetNfpRomInfo to get the information in the shared region to determine whether the NFP tag is compatible with the application.

> **Warning:** If the tag was mounted with nn::nfp::MountRom, the nn::nfp::GetNfpCommonInfo() function returns nn::nfp::ResultInvalidOperation.

The shared region information retrieved by `nn::nfp::GetNfpCommonInfo` is defined with the `nn::nfp::CommonInfo` structure. A pointer to this structure must be passed to the function.

The shared region information retrieved by `nn::nfp::GetNfpRomInfo` is defined with the `nn::nfp::RomInfo` structure. A pointer to this structure must be passed to the function.

Only use the character ID (`characterId`) to determine compatibility of the NFP tag. You must prepare a list of compatible character IDs in your application.

**Warning:** Using any information other than the character ID in an application is currently prohibited.

**Table 6-14. Possible Return Values From nn::nfp::GetNfpCommonInfo**

| Return Value | Description |
|---|---|
| `nn::ResultSuccess` | The information was successfully retrieved. |
| `nn::nfp::ResultInvalidPointer` | The pointer passed as an argument is invalid. |
| `nn::nfp::ResultInvalidOperation` | The library is not in an appropriate state for calling this function. |
| `nn::nfp::ResultSleep` | The process failed because the system is entering Sleep Mode. |
| `nn::nfp::ResultWifiOff` | The process failed because the system is in wireless-disabled mode. |

**Table 6-15. Possible Return Values From nn::nfp::GetNfpRomInfo**

| Return Value | Description |
|---|---|
| `nn::ResultSuccess` | The information was successfully retrieved. |
| `nn::nfp::ResultInvalidPointer` | The pointer passed as an argument is invalid. |
| `nn::nfp::ResultInvalidOperation` | The library is not in an appropriate state for calling this function. |
| `nn::nfp::ResultSleep` | The process failed because the system is entering Sleep Mode. |
| `nn::nfp::ResultWifiOff` | The process failed because the system is in wireless-disabled mode. |

The shared region includes registration information that is retrieved using the `nn::nfp::GetNfpRegisterInfo()` function in addition to the information that is retrieved using `nn::nfp:GetNfpCommonInfo()` and `nn::nfp::GetNfpRomInfo()`.

**Warning:** If the tag was mounted with `nn::nfp::MountRom`, the `nn::nfp::GetNfpRegisterInfo()` function returns `nn::nfp::ResultInvalidOperation`.

Registration information is defined with the `nn::nfp::RegisterInfo` structure. A pointer to the structure must be passed to `nn::nfp::GetNfpRegisterInfo`.

If a value of `nn::nfp::ResultNeedRegister` is returned, an owner has not yet been registered for the NFP tag in **amiibo Settings**. Start **amiibo Settings** if your application requires pairing information.

**Note:** For more information, see 6.5. Starting amiibo Settings.

The registration information includes data about the Mii that is registered as the owner (`miiData`), the nickname of the amiibo (`nickName`), and the font region required to display the nickname (`fontRegion`).

To use the Mii data inside your application, you must include the Face Library. For more information about how to display Mii characters in your application, see the Face Library documentation.

Nicknames are stored using UTF-16 BE, so convert the endianness and encoding as required. The terminating character is NULL (`0x0000`). When displaying a nickname in your application, handle the display of unsupported characters in a way that does not corrupt the display or prevent the application from proceeding. For example, you can use the system font specified in the font region or display substitute characters for characters that are not included in the application font.

If an owner is registered, Mii data is definitely available, but in some cases a nickname might not be set (it might be an empty string). If your application requires a nickname, start **amiibo Settings** using the same procedure described for `nn::nfp::ResultNeedRegister`.

> **Note:** Applications that display the nickname must take care in the handling of amiibo nicknames and Mii nicknames. In particular, avoid displaying the nickname in a manner that implies it is the name of the owner Mii. Also make sure that you display a substitute string such as "(owner name)'s amiibo," "(amiibo character name)," or just "amiibo" if the nickname is not set.

> **Warning:** Using country information (`country`) in an application is currently prohibited.

**Table 6-16. Possible Return Values From nn::nfp::GetNfpRegisterInfo**

| Return Value | Description |
|---|---|
| `nn::ResultSuccess` | The information was successfully retrieved. |
| `nn::nfp::ResultInvalidPointer` | The pointer passed as an argument is invalid. |
| `nn::nfp::ResultInvalidOperation` | The library is not in an appropriate state for calling this function. |
| `nn::nfp::ResultSleep` | The process failed because the system is entering Sleep Mode. |
| `nn::nfp::ResultWifiOff` | The process failed because the system is in wireless-disabled mode. |
| `nn::nfp::ResultNeedRegister` | The owner has not been registered in **amiibo Settings**. |

## 6.6.2. Accessing the Application-Specific Region

Access privileges for the application-specific region are controlled by the access ID. Before opening access to the application-specific region, you must first call `nn::nfp::OpenApplicationArea()` and check whether the application has access privileges.

If a value of `nn::nfp::ResultAccessIdMisMatch` is returned, the application does not have access privileges because the application-specific area was created by a different access ID. In this case, both reading and writing are restricted. There is no way to delete the application-specific region from an application. If you want to use this region for your application, you must use **amiibo Settings** to delete it.

A return value of `nn::nfp::ResultNeedCreate` means the mounted NFP tag does not have an application-specific region. Your application must create one.

**Table 6-17. Possible Return Values From nn::nfp::OpenApplicationArea**

| Return Value | Description |
|---|---|
| `nn::ResultSuccess` | The process was successful. The application-specific region can be accessed. |
| `nn::nfp::ResultInvalidOperation` | The library is not in an appropriate state for calling this function. |
| `nn::nfp::ResultSleep` | The process failed because the system is entering Sleep Mode. |
| `nn::nfp::ResultWifiOff` | The process failed because the system is in wireless-disabled mode. |
| `nn::nfp::ResultAccessIdMisMatch` | The application-specific area cannot be accessed because it was created by a different access ID. |
| `nn::nfp::ResultNeedCreate` | The application-specific region has not been created. |

### 6.6.2.1. Creating the Application-Specific Region

The application-specific region is created by calling the `nn::nfp::CreateApplicationArea()` function.

Use the `nn::nfp::InitializeCreateInfo()` function to initialize the `nn::nfp::ApplicationAreaCreateInfo` structure that is passed to this function before

setting the member variables.

If the data written to the application-specific region will also be read by Cafe, we recommend standardizing the byte order because the Cafe and CTR/SNAKE architectures have different endianness.

> **Note:** Creating the application-specific region can take longer than a second.
>
> If you want to try the operation again, we recommend a retry interval of around 100 milliseconds.
>
> Because the function performs the same operations as `nn::nfp::Flush`, you do not need to call `nn::nfp::Flush` after calling `nn::nfp::CreateApplication`.

**Table 6-18. Possible Return Values From nn::nfp::CreateApplicationArea**

| Return Value | Description |
|---|---|
| `nn::ResultSuccess` | The process was successful. The application-specific region has been created. |
| `nn::nfp::ResultInvalidArgument` | The process failed due to a problem in the argument that was specified. This result is an implementation error that must be fixed in the development phase. |
| `nn::nfp::ResultInvalidPointer` | The process failed due to a problem in the argument that was specified. This result is an implementation error that must be fixed in the development phase. |
| `nn::nfp::ResultInvalidOperation` | The library is not in an appropriate state for calling this function. |
| `nn::nfp::ResultSleep` | The process failed because the system is entering Sleep Mode. |
| `nn::nfp::ResultWifiOff` | The process failed because the system is in wireless-disabled mode. |
| `nn::nfp::ResultNotSupported` | The tag is not supported. The tag may have been switched with a different tag. |
| `nn::nfp::ResultAlreadyCreated` | The application-specific region has already been created. |
| `nn::nfp::ResultTagNotFound` | The tag could not be found. This result is caused by the tag leaving the range of communication or being swapped with a different tag. |
| `nn::nfp::ResultNeedRetry` | An error occurred during writing. The process might succeed if tried again. |
| `nn::nfp::ResultOperationFailed` | An error occurred during writing. The tag content may be corrupted. |
| `nn::nfp::ResultConnectCanceled` | The NFC Reader/Writer was disconnected due to an internal error. |

The following sample code retries the operation up to three times.

**Code 6-1. Creating the Application-Specific Region**

```
#define MY_ACCESS_ID 0x12345678
#define MY_APPLICATION_AREA_SIZE 128
nn::Result result;
bit8 writeData[MY_APPLICATION_AREA_SIZE];

// First, initialize the settings information.
nn::nfp::ApplicationAreaCreateInfo createInfo;
result = nn::nfp::InitializeCreateInfo(&createInfo);
if (result.IsFailure())
{
    // Error.
    return;
}

// Prepare the initialization data.
{
    memset(writeData, NULL, sizeof(writeData));
    // Write the data with big-endian byte order.
    writeData[0] = 'I';
    writeData[1] = 'N';
    writeData[2] = 'I';
    writeData[3] = 'T';
    writeData[4] = 0x00;
    writeData[5] = 0x00;
    writeData[6] = 0x00;
    writeData[7] = 0x01;
}
// Set the data required for initialization.
createInfo.accessId = MY_ACCESS_ID;
createInfo.pInitialData = writeData;
createInfo.initialDataSize = sizeof(writeData);
```

```
// Create the application-specific region.
int retryCount = 0;
while (true)
{
    result = nn::nfp::CreateApplicationArea(createInfo);
    if (result.IsSuccess()) break;
    if (result.IsFailure())
    {
        if (nn::nfp::ResultNeedRetry::Includes(result))
        {
            // Retry up to 3 times.
            retryCount++;
            if (retryCount < 3)
            {
                // Wait 0.1 seconds before retrying.
                nn::os::Thread::Sleep(100);
                continue;
            }
            // Process failed. Handle by starting over from tag detection or a
similar measure.
        }
        else if (nn::nfp::ResultAlreadyCreated::Includes(result))
        {
            // The application-specific region has already been created.
            // Make sure the data is valid. The process succeeded if no
problems are found.
        }
        else if (nn::nfp::ResultNotSupported::Includes(result))
        {
            // The tag might have been switched.
        }
        else if (nn::nfp::ResultOperationFailed::Includes(result))
        {
            // The tag might be corrupted.
        }
        else if (nn::nfp::ResultTagNotFound::Includes(result))
        {
            // The tag might have been removed from the reader.
        }
        else if (nn::nfp::ResultInvalidOperation::Includes(result))
        {
            // Entered a state where the function cannot be called.
        }
        else if (nn::nfp::ResultWifiOff::Includes(result))
        {
            // Failed because the system is in wireless-disabled mode.
        }
        else if (nn::nfp::ResultSleep::Includes(result))
        {
            // Failed because the system is in Sleep Mode.
         }
        else if (nn::nfp::ResultConnectCanceled::Includes(result))
        {
            // Failed because NFC Reader/Writer was disconnected
        }
        else
        {
            // Other error
        }
        return;
    }
}
// nn::nfp::Flush does not need to be called.
```

### 6.6.2.2. Reading the Application-Specific Region

To read the data that is written to the application-specific region, call the `nn::nfp::ReadApplicationArea()` function.

This allows you to read the specified number of bytes from the beginning of the application-specific region. The maximum size of the data that can be read from the application-specific region can be retrieved from the `applicationAreaSize` member of the `CommonInfo` structure that can be retrieved using the `nn::nfp::GetCommonInfo()` function.

**Table 6-19. Possible Return Values From nn::nfp::ReadApplicationArea**

| Return Value | Description |
|---|---|
| `nn::ResultSuccess` | The process was successful. |
| `nn::nfp::ResultInvalidArgument` | The process failed due to a problem in the argument that was specified. This result is an implementation error that must be fixed in the development phase. |
| `nn::nfp::ResultInvalidPointer` | The process failed due to a problem in the argument that was specified. This result is an implementation error that must be fixed in the development phase. |
| `nn::nfp::ResultInvalidOperation` | The library is not in an appropriate state for calling this function. |
| | The process failed because the system is entering Sleep |

| | |
|---|---|
| nn::nfp::ResultSleep | Mode. |
| nn::nfp::ResultWifiOff | The process failed because the system is in wireless-disabled mode. |

### 6.6.2.3. Writing to the Application-Specific Region

The application-specific region is written to by calling the nn::nfp::WriteApplicationArea() function, followed by the nn::nfp::Flush() function.

The UID of the target NFP tag is required when calling nn::nfp::WriteApplicationArea. Take the value of the tagId member of the TagInfo structure that was retrieved using nn::nfp::GetTagInfo beforehand and pass it as an argument. We recommend encoding the data to be written to the application-specific region using the same big-endian encoding as the information written to the tag by the NFP library.

The data is not actually written to the application-specific region when nn::nfp::WriteApplicationArea finishes executing. It is stored in a cache in the NFP library. The data is actually written to the application-specific region of the NFP tag when the nn::nfp::Flush() function is called.

> **Note:** Writing to the application-specific region of the NFP tag can take longer than a second.
>
> If you want to try the operation again, we recommend a retry interval of around 100 milliseconds.

**Table 6-20. Possible Return Values From nn::nfp::WriteApplicationArea**

| Return Value | Description |
|---|---|
| nn::ResultSuccess | The process was successful. |
| nn::nfp::ResultInvalidArgument | The process failed due to a problem in the argument that was specified. This result is an implementation error that must be fixed in the development phase. |
| nn::nfp::ResultInvalidPointer | The process failed due to a problem in the argument that was specified. This result is an implementation error that must be fixed in the development phase. |
| nn::nfp::ResultInvalidOperation | The library is not in an appropriate state for calling this function. |
| nn::nfp::ResultSleep | The process failed because the system is entering Sleep Mode. |
| nn::nfp::ResultWifiOff | The process failed because the system is in wireless-disabled mode. |
| nn::nfp::ResultUidMisMatch | The tag UID does not match the specified UID. |

**Table 6-21. Possible Return Values From nn::nfp::Flush**

| Return Value | Description |
|---|---|
| nn::ResultSuccess | The process was successful. |
| nn::nfp::ResultInvalidOperation | The library is not in an appropriate state for calling this function. |
| nn::nfp::ResultSleep | The process failed because the system is entering Sleep Mode. |
| nn::nfp::ResultWifiOff | The process failed because the system is in wireless-disabled mode. |
| nn::nfp::ResultNotSupported | The tag is not supported. The tag may have been switched with a different tag. |
| nn::nfp::ResultTagNotFound | The tag could not be found. This result is caused by the tag leaving the range of communication or being swapped with a different tag. |
| nn::nfp::ResultOperationFailed | An error occurred during writing. The tag content may be corrupted. |
| nn::nfp::ResultNeedRetry | An error occurred during writing. The process might succeed if tried again. |
| nn::nfp::ResultConnectCanceled | The NFC Reader/Writer was disconnected due to an internal error. |

## 6.7. Unmounting a Tag

Tags are unmounted by calling the `nn::nfp::Unmount()` function.

**Table 6-22. Possible Return Values From nn::nfp::Unmount**

| Return Value | Description |
|---|---|
| `nn::ResultSuccess` | The process was successful. |
| `nn::nfp::ResultInvalidOperation` | The library is not in an appropriate state for calling this function. |
| `nn::nfp::ResultSleep` | The process failed because the system is entering Sleep Mode. |
| `nn::nfp::ResultWifiOff` | The process failed because the system is in wireless-disabled mode. |

## 6.8. Stopping Tag Detection

Call the `nn::nfp::StopDetection()` function to stop tag detection.

If this function is called while a tag is mounted, the NFP library unmounts the tag.

If the tag is in a detected state when `nn::nfp::StopDetection` is called, the application receives a tag loss notification.

**Table 6-23. Possible Return Values From nn::nfp::StopDetection**

| Return Value | Description |
|---|---|
| `nn::ResultSuccess` | The process was successful. |
| `nn::nfp::ResultInvalidOperation` | The library is not in an appropriate state for calling this function. |
| `nn::nfp::ResultSleep` | The process failed because the system is entering Sleep Mode. |
| `nn::nfp::ResultWifiOff` | The process failed because the system is in wireless-disabled mode. |
| `nn::nfp::ResultConnectCanceled` | The NFC Reader/Writer was disconnected due to an internal error. |
| `nn::nfp::ResultIrFunctionError` | The IrDA module may have been damaged. |

## 6.9. Finalization

Call the `nn::nfp::Finalize()` function to finalize the NFP library. This function unmounts the tag, stops tag detection, and, if a tag had been detected, issues a tag loss notification.

**Table 6-24. Possible Return Values From nn::nfp::Finalize**

| Return Value | Description |
|---|---|
| `nn::ResultSuccess` | The process was successful. |
| `nn::nfp::ResultInvalidOperation` | The library has already been finalized. |

## 6.10. Using in 3DS Applications

To use the NFP library in a 3DS application, you must implement code to support various 3DS features such as the HOME Button, Sleep Mode, and the POWER Button.

In addition, the NFC Reader/Writer for CTR uses the infrared port on the CTR system. This

necessitates the handling of notifications from the reader and when the reader is connected and disconnected.

A problem that affects both SNAKE and CTR is that NFP library operation develops problems when the system is under a load.

## 6.10.1. Handling the HOME Button

When a HOME Button press is detected, call `nn::nfp::Finalize` to put the NFP library into the `NONE` state before transitioning to the HOME Menu. The same operation is required before starting an applet. If you forget to do this, the system automatically puts the NFP library in NONE status in the Release build. However, an assertion occurs in the Debug or Development build and causes the program to stop operating.

You can display the HOME Menu Disabled icon and not transition to the HOME Menu if it is necessary to protect the integrity of the tag data because, for example, the application is in the middle of writing data to the application-specific region. Data writing usually finishes in about one to two seconds.

**Code 6-2. Handling the HOME Button**

```
// Check if transitioning to the HOME Menu.
if (nn::applet::IsExpectedToProcessHomeButton())
{
    // Do not transition to the HOME Menu if writing to the NFP tag.
    if (_isNfpTagWritingInProgress())
    {
        // Display the HOME Menu Disabled icon.
        _displyHomeNixSign();
        nn::applet::ClearHomeButtonState();
    }
    else
    {
        // Stop access to the NFP tag.
        // Call nn::nfp::Finalize to enter the NONE state before transitioning
to the HOME Menu.
        _stopNfpTagAccess();
        _finalizeNfpTagAccess();
        // Processes to run before transitioning to the HOME Menu.
        _prepareHomeMenu();
        // Transition to the HOME Menu.
        nn::applet::ProcessHomeButtonAndWait();
        if (nn::applet::IsExpectedToCloseApplication())
        {
            // Close the application.
  _closeApplication();
        }
        // Processes after returning from the HOME Menu.
        _returnFromHomeMenu();
        // Start tag detection here if resuming immediately. (Call
nn::nfp::StartDetection.)
        if (_needRestartNfpTagAccess()) _startNfpTagAccess();
    }
}
```

## 6.10.2. Handling Sleep Mode Requests

The NFP library automatically changes the state to `INIT` when the system transitions to Sleep Mode. However, we recommend having the application handle the transition if it is necessary to protect the integrity of the NFP tag data because, for example, the application is in the middle of writing to the application-specific region.

**Code 6-3. Handling Sleep Mode Requests**

```
// Handling Sleep Mode Requests
// Assumes the sleep query callback function will wait for the reply.
if (nn::applet::IsExpectedToReplySleepQuery())
{
    // Check whether the NFP tag is being written to.
    if (_isNfpTagWritingInProgress())
    {
        // Do not enter Sleep Mode if the NFP tag is being written to.
    }
    else
    {
        // Processes to run before Sleep Mode.
        _prepareSleep();
        // Allow transition to Sleep Mode.
        nn::applet::ReplySleepQuery(nn::applet::REPLY_ACCEPT);
        // Wait for the event to be signaled by the wakeup callback, etc.
        _waitAwakeEvent();
```

```
        // Processes for returning from Sleep Mode.
        _returnFromSleep();
        // Start tag detection here if resuming immediately. (Call
nn::nfp::StartDetection.)
        if (_needRestartNfpTagAccess()) _startNfpTagAccess();
    }
}
```

## 6.10.3. Handling the POWER Button

When the POWER Button is pressed, the application must quickly display the POWER Menu. Before transitioning to the HOME Menu or an applet, call `nn::nfp::Finalize` to put the NFP library into the NONE state. If you forget to do this, the system automatically puts the NFP library in NONE status in the Release build. However, an assertion occurs in the Debug or Development build and causes the program to stop operating.

However, the application can wait for the writing process to complete before displaying the POWER Menu if it is necessary to protect the integrity of the NFP tag data because, for example, the application is in the middle of writing data to the application-specific region. Data writing usually finishes in about one to two seconds.

**Code 6-4. Handling the POWER Button**

```
// Handling the POWER Button
if (nn::applet::IsExpectedToProcessPowerButton())
{
    // Check whether the NFP tag is being written to.
    if (_isNfpTagWritingInProgress())
    {
        // Do not handle the POWER Button until finished writing to the NFP tag.
    }
    else
    {
        // Stop access to the NFP tag.
          // Before transitioning to the POWER menu, nn::nfp::Finalize must be
called to
        // transition to the NONE state.
        _stopNfpTagAccess();
        _finalizeNfpTagAccess();
        // Processes to run before transitioning to the POWER Menu.
        _preparePowerButtonMenu();
        // Display the POWER Menu.
        nn::applet::ProcessPowerButtonAndWait();
        if (nn::applet::IsExpectedCloseApplication())
        {
            // Application finalization process.
                    _closeApplication();
        }
        else
        {
            // Return from the POWER Menu.
            _returnFromPowerButtonMenu();
        }
    }
}
```

## 6.10.4. Implementation Requirements for CTR

If your application uses the NFC Reader/Writer for CTR, you must explicitly connect to the device from the application. Your application also must handle the connected and disconnected states of the device, in addition to the tag detection and loss notifications.

The following four functions are related to using the NFC Reader/Writer

- `nn::nfp::Connect`
- `nn::nfp::GetTargetConnectionStatus`
- `nn::nfp::GetConnectResult`
- `nn::nfp::Disconnect`

The `nn::nfp::Connect()` function requests that the NFP library establish a connection to the reader. This function is asynchronous. The `nn::nfp::GetTargetConnectionStatus()` function returns a value `nn::nfp::TARGET_CONNECTED` if the connection succeeded or `nn::nfp::TARGET_DISCONNECTED` if the connection failed.

If the NFC reader/write and CTR are correctly oriented, the connection process takes several hundred milliseconds to complete. However, the connection can take a full second to complete if

the system attempts to connect immediately after the NFC Reader/Writer is turned on due to an internal initialization process. If the NFC Reader/Writer and CTR system are not correctly oriented, the system attempts to connect for one second until it times out, after which the state becomes `nn::nfp::TARGET_DISCONNECTED` .

If the `nn::nfp::GetTargetConnectionStatus()` function is called on SNAKE hardware, it always returns `nn::nfp::TARGET_CONNECTED`.

**Table 6-25. Values Obtained With nn::nfp::GetTargetConnectionStatus**

| Obtained Value | Connection to NFC Reader/Writer |
|---|---|
| `nn::nfp::TARGET_DISCONNECTED` | Disconnected |
| `nn::nfp::TARGET_CONNECTING` | Connected |
| `nn::nfp::TARGET_CONNECTED` | Connected |

**Note:** Consequently, if the CTR is pointed away from the NFC Reader/Writer while the user is playing, the disconnect is detected in a relatively short amount of time.
If you do not want the user to be aware of the disconnection, you can implement in the application to keep track of the connection state and automatically call `nn::nfp::Connect` if the state is `nn::nfp::TARGET_DISCONNECTED`.

If the connection fails, the `nn::nfp::GetConnectResult()` function is called to get the cause of the failure. Call this function after the `nn::nfp::GetTargetConnectionStatus()` function returns `nn::nfp::TARGET_DISCONNECTED`.

If the `nn::nfp::GetConnectResult()` function is called on SNAKE hardware, it always returns `nn::ResultSuccess`.

**Table 6-26. Values Acquired With nn::nfp::GetConnectResult**

| Get Failure Cause | Description |
|---|---|
| `nn::ResultSuccess` | Connection succeeded, or a disconnect occurred unrelated to calling `Connect`. |
| `nn::nfp::ResultConnectCanceled` | Connection processing was canceled, or failed due to an internal error. |
| `nn::nfp::ResultTimeOutError` | The process timed out without successful connection. |
| `nn::nfp::ResultUpdateRequired` | NFC Reader/Writer must be updated. |
| `nn::nfp::ResultIrFunctionError` | The IrDA module may have been damaged. |
| `nn::nfp::ResultNfcTargetError` | The NFC Reader/Writer may have been damaged. |

**Note:** For information about error handling when connection to the NFC Reader/Writer has failed, see 5.11.4. NFC Reader/Writer Connection Sequence.

The `nn::nfp::Disconnect()` function is called by the application when you explicitly want to disconnect from the reader. This process takes between 10 and 100 milliseconds to complete. If tag detection is currently in progress (`nn::nfp::StartDetection` was called), the `nn::nfp::StopDetection()` function is called internally to stop tag detection. The entire process takes around 100 milliseconds to complete.

This function returns `nn::nfp::ResultInvalidOperation` if called when the NFC Reader/Writer is already disconnected. We recommend calling the `nn::nfp::GetTargetConnectionStatus()` function to accurately keep track of the connection status to the NFC Reader/Writer.

When `nn::nfp::Disconnect` is called on SNAKE, only `nn::nfp::StopDetection` is executed internally to stop tag detection.

**Note:** The remaining battery life of the NFC Reader/Writer is only checked once, immediately after it is connected to the CTR using the `nn::nfp::Connect()` function. It is not checked again until the device is disconnected.
Consequently, if the NFC Reader/Writer continuously detects and writes tags for several minutes without updating the connection state, it might suddenly turn off without the red remaining battery life warning LED turning on.

If your application continuously detects and writes tags, we recommend calling the `nn::nfp::Disconnect and nn::nfp::Connect()` functions on a regular basis (for

example, once per minute).

For some battery types, the LED might light up blue when the power is turned back on after the device loses power as described above. This is by design.
If you continue to use the NFC Reader/Writer without changing the batteries, the LED will turn red before the device loses power again.

The following sample code shows how to start and stop tag detection.

**Code 6-5. Starting and Stopping Tag Detection**

```
bool _willTerminate = false;
bool _connected = false;
nn::Result _lastResult = nn::ResultSuccess();
nn::nfp::TargetConnectionStatus _targetConnectionStatus;

// Tag detection notification event, set in nn::nfp::SetActivateEvent()
nn::os::Event _eventActivate;

// Tag loss notification event, set in nn::nfp::SetDeactivateEvent()
nn::os::Event _eventDeactivate;

// Start tag detection
nn::Result _startDetection()
{
    // CTR-specific implementation
    if (!nn::os::IsRunOnSnake())
    {
        // External NFC connection
        if (!_connected)
        {
            _lastResult = nn::nfp::Connect();
            if (_lastResult.IsSuccess())
            {
                // Wait for connection processing to complete.
                while (1)
                {

nn::os::Thread::Sleep(nn::fnd::TimeSpan::FromMilliSeconds(500));

nn::nfp::GetTargetConnectionStatus(&_targetConnectionStatus);
                    if (_targetConnectionStatus == nn::nfp::TARGET_CONNECTED)
                    {
                        // Successful connection
                        _connected = true;
                        break;
                    }
                    else if (_targetConnectionStatus ==
nn::nfp::TARGET_DISCONNECTED)
                    {
                        // Connection failed or connection timeout
                        _connected = false;
                        nn::nfp::GetConnectResult(&_lastResult);
                        return _lastResult;
                    }
                }
            }
            else
            {
                if (nn::nfp::ResultInvalidOperation::Includes(_lastResult))
                {
                    // Abnormal state
                }
                else
                {
                    // Other errors stopped with PANIC
                    NN_PANIC_WITH_RESULT(_lastResult);
                }
                return _lastResult;
            }
        }
    }

    // Start detection
    _lastResult = nn::nfp::StartDetection();
    if (_lastResult.IsSuccess())
    {
        // Processing success
        // Prompt the user to touch the tag to the touchpoint.
    }
    else if (nn::nfp::ResultWifiOff::Includes(_lastResult))
    {
        // Wireless off mode
    }
    else if (nn::nfp::ResultSleep::Includes(_lastResult))
    {
        // Sleep Mode
    }
    else if (nn::nfp::ResultInvalidOperation::Includes(_lastResult))
    {
        // Abnormal state
    }
    else
    {
        // Other errors stopped with PANIC
        NN_PANIC_WITH_RESULT(_lastResult);
    }
    return _lastResult;
```

```
}

// Stop tag detection
nn::Result _stopDetection(bool forceDisconnect = false)
{
    // Stop detection
    _lastResult = nn::nfp::StopDetection();
    if (_lastResult.IsSuccess())
    {
        // Processing success
    }
    else if (nn::nfp::ResultWifiOff::Includes(_lastResult))
    {
        // Wireless off mode
    }
    else if (nn::nfp::ResultSleep::Includes(_lastResult))
    {
        // Sleep Mode
    }
    else if (nn::nfp::ResultInvalidOperation::Includes(_lastResult))
    {
        // Abnormal state
    }
    else
    {
        // Other errors stopped with PANIC
        NN_PANIC_WITH_RESULT(_lastResult);
    }

    // CTR-specific implementation
    if (!nn::os::IsRunOnSnake())
    {
        // External NFC disconnected
        if (forceDisconnect && _connected)
        {
            _connected = false;
            _lastResult = nn::nfp::Disconnect();
            if (_lastResult.IsSuccess())
            {
                // Processing success
            }
            else if (nn::nfp::ResultInvalidOperation::Includes(_lastResult))
            {
                // Abnormal state
            }
            else
            {
                // Other errors stopped with PANIC
                NN_PANIC_WITH_RESULT(_lastResult);
            }
        }
    }
    return _lastResult;
}

// Thread function waiting for notification from the NFP library
void _procTagDetection()
{
    const int waitMilliSeconds = 500;
    nn::os::WaitObject* events[2] = {
        &_eventActivate, &_eventDeactivate };
    while (!_willTerminate)
    {
        const int eventIndex = nn::os::WaitObject::WaitAny(
            events, 2, nn::fnd::TimeSpan::FromMilliSeconds(waitMilliSeconds));
        if (eventIndex == 0)
        {
            _activate();
        }
        else if (eventIndex == 1)
        {
            _deactivate();
        }
        else
        {
            nn::nfp::TargetConnectionStatus currentTargetConnectionStatus;
            nn::nfp::GetTargetConnectionStatus(&currentTargetConnectionStatus);
            if( _targetConnectionStatus != currentTargetConnectionStatus )
            {
                if (currentTargetConnectionStatus == nn::nfp::TARGET_CONNECTED)
                {
                    _connected = true;
                }
                else if (currentTargetConnectionStatus ==
nn::nfp::TARGET_DISCONNECTED)
                {
                    _connected = false;
                }
                _targetConnectionStatus = currentTargetConnectionStatus;
            }
        }
    }
}
```

## 6.10.5. NFP Library Operation When the System Is Under Load

The following kinds of problems will develop when using the NFP library while the system is under a high processing load.

- Reading and writing of NFP tags slows.
- The possibility of NFP tag writing errors increases.
- Disconnection from NFC Reader/Writer.
  - NFC Reader/Writer disconnection becomes more likely when a tag is detected. This pertains to the NFP library states `RW_ACTIVE`, `RW_MOUNT`, and `RW_MOUNT_ROM`.

Because of these problems, note the following points when developing applications.

1. Do not make needless use of the following devices.

- Camera, YUV/RGB converter, microphone, accelerometer, gyroscope, wireless connection, DSP, debug pad. (See Note 1.)
- Be particularly careful with the following devices, because using them significantly increases the load on the system core.
  - Microphone (when used at a high sampling rate), accelerometer, wireless, and debug pad.

2. Do not assign the system core to the application for needlessly long periods of time.
By reducing the amount of time the system core is assigned to the application, you can ease the problems caused by the use of these devices. For more information about using the system core, see Processing Handled by the System Core in *3DS Performance Tips*.

3. Reduce the number of issued graphics draw commands (3D commands and command requests). You can lessen the system load by issuing fewer draw commands and, in addition, applying the tips presented in the Graphics Processing section of *3DS Performance Tips*.

Depending on the extent to which the load on the system is increased by the use of the NFP library, the result can be a slowdown in graphics processing and the generation of audio noise. These symptoms can be alleviated by stopping the listed devices.

**Note 1**: Simply connecting the debug pad to the debugger can increase the system load, even when it is not being used with the application.

For reference purposes, the following sample program based on `demo1` incorporates features that use the NFP library and features that use local communication, the gyro sensor and the DSP, and is implemented in such a way that the NFC Reader/Writer frequently disconnects when used with the CTR.

In this demo, the NFP library is used to only to transition between the RW_ACTIVE and RW_MOUNT states after tags are detected. The result is that it is always likely for the NFC Reader/Writer to become disconnected.

| Enabled Devices (and Settings Used) | Disabled Devices | Frequency of Disconnection (per Hour) |
|---|---|---|
| DSP (aac), local communication (two units communicating at 1 Mbps) | Camera, YUV/RGB converter, microphone, accelerometer, gyroscope, debug pad. | 0.25 times |
| Microphone (SAMPLING_RATE_32730), DSP (aacdec), local communication (two units communicating at 1 Mbps) | Camera, YUV/RGB converter, accelerometer, gyroscope, debug pad. | 1 time |
| Camera (FRAME_RATE_30), DSP (aacdec), accelerometer, gyroscope, local communication (two units communicating at 1 Mbps) | Microphone, debug pad. | 0 times |
| Microphone (SAMPLING_RATE_32730), camera (FRAME_RATE_30), DSP (aacdec), accelerometer, gyroscope. | Local communication, debug pad. | 1.5 times |
| Microphone (SAMPLING_RATE_8180), camera (FRAME_RATE_30), DSP (aacdec), accelerometer, gyroscope. | Local communication, debug pad. | 0 times |

Note that the load placed on the system core in `demo1` is relatively light compared to a normal application. For actual application development, note that disconnections may occur more frequently.

# Revision History

## Version 1.4 (2016-05-10)

### Changes

- 5.1. Initialization Sequence
    - Added the `ResultIrFunctionError` error value returned by `Initialize()` in the Initialization Sequence flow.

## Version 1.3 (2015-11-05)

### Changes

- 1. Overview
    - Added an instruction to end any other infrared communication functions in advance if they are being used.
- 5.7. Tag Mounting Sequence
    - Deleted `ResultTagNotFound` from possible return values from `nn::nfp::GetTagInfo()`.
- 5.8. Tag Restoration Sequence
    - Deleted `ResultTagNotFound` from possible return values from `nn::nfp::GetTagInfo()`.
- 5.10.5. Application-Specific Region Check Sequence
    - Deleted `ResultTagNotFound` from possible return values from `nn::nfp::GetTagInfo()`.
- 6. Using the NFP Library
    - Deleted notes about not letting the state become NONE because this is no longer required due to the SDK revision.
- 6.2. Starting Tag Detection
    - Added information about the restrictions on high-frequency tag detection.
- 6.3. Getting Tag Information
    - Deleted `ResultTagNotFound` from possible return values from `nn::nfp::GetTagInfo()`.
- 6.10.4. Implementation Requirements for CTR
    - Fixed an error in sample code.

# Version 1.2 (2015-04-28)

## Additions

- 6.10.5. NFP Library Operation When the System Is Under Load

## Changes

- 2. NFP Library Features
    - Added that, for tags other than amiibo, there are differences between tags that can detect Nintendo 3DS and New Nintendo 3DS.
    - Added that two tags cannot be used at the same time, and that problems arise if you try to do so.
- 4. State Transitions
    - Revised information about battery life when using the NFC Reader/Writer.
- 5.3. Common Error Handling
    - Added `nn::nfp::ResultConnectCanceled` to the common error handling.
- 5.8. Tag Restoration Sequence
    - Added `nn::nfp::ResultConnectCanceled` to the `nn::nfp::Restore` errors.
- 5.10.1. Common Information Retrieval Sequence
    - Deleted `nn::nfp::ResultTagNotFound` from the `nn::nfp::GetNfpCommonInfo` errors.
- 5.10.2. Registration Information Retrieval Sequence
    - Deleted `nn::nfp::ResultTagNotFound` from the `nn::nfp::GetNfpRegisterInfo` errors.
- 5.10.4. Application-Specific Region Creation Sequence
    - Added `nn::nfp::ResultConnectCanceled` to the `nn::nfp::CreateApplicationArea` errors.
- 5.10.6. Application-Specific Region Read Sequence
    - Deleted `nn::nfp::ResultTagNotFound` from the `nn::nfp::ReadApplicationArea` errors.
- 5.10.7. Application-Specific Region Write Sequence
    - Added `nn::nfp::ResultConnectCanceled` to the `nn::nfp::Flush` errors.
- 5.11.3. Power Off Sequence
    - Noted that the NFP library must be exited before displaying the POWER Menu.
- 5.11.4. NFC Reader/Writer Connection Sequence
    - Added information about errors due to transitions to Sleep Mode and Wireless-disabled

mode.
    - Added handling for `ResultConnectCanceled` and `ResultTimeOutError`.
    - Deleted `ResultInvalidOperation` from the causes of failures in `nn::nfp::GetConnectResult` in the flowchart.

- 6. Using the NFP Library

    - Added cautions about using the NFP library.
    - Corrected NFP library function blocking times.

- 6.1. Initialization

    - Corrected NFP library function blocking times.
    - Added `nn::nfp::ResultIrFunctionError` to the values returned by `nn::nfp::Initialize`.

- 6.2. Starting Tag Detection

    - Corrected the times required for tag discovery and forfeiture.

- 6.4. Mounting a Tag

    - Corrected NFP library function blocking times.

- 6.4.1. Restoring Tags

    - Added `nn::nfp::ResultConnectCanceled` to the values returned by `nn::nfp::Restore`.

- 6.6.2.1. Creating the Application-Specific Region

    - Added `nn::nfp::ResultConnectCanceled` to the values returned by `nn::nfp::CreateApplicationArea`.

- 6.6.2.2. Reading the Application-Specific Region

    - Added `nn::nfp::ResultSleep` and `nn::nfp::ResultWifiOff` to the values returned by `nn::nfp::ReadApplicationArea()`.

- 6.6.2.3. Writing to the Application-Specific Region

    - Added `nn::nfp::ResultConnectCanceled` and `nn::nfp::ResultOperationFailed` to the values returned by `nn::nfp::Flush()`.

- 6.8. Stopping Tag Detection

    - Added `nn::nfp::ResultConnectCanceled` and `nn::nfp::ResultIrFunctionError` to the values returned by `nn::nfp::StopDetection()`.

- 6.10.1. Handling the HOME Button

    - Added a description of the behavior when transitioning to the HOME Menu without exiting the NFP library.
    - Noted sample code for exiting the NFP library when transitioning to the HOME Menu.

- 6.10.4. Implementation Requirements for CTR

    - Added a list of values obtained when acquiring the connection status.
    - Added a list of values obtained from the causes of failures.
    - Added a note to refer to the NFC Reader/Writer Connection Sequence for information about error handling.

# Version 1.1 (2015-01-15)

## Additions

- 5.11.4. NFC Reader/Writer Connection Sequence

## Changes

- 1. Overview
  - Noted that the error code list includes sample error messages.
- 4. State Transitions
  - Added a note that the battery life when using the NFC Reader/Writer is currently being evaluated.
  - Added information about the SNAKE battery life.
  - Added information about the state transitions when the NFC Reader/Writer is disconnected.
- 5.3. Common Error Handling
  - Removed the comment about needing to notify the user when a `ResultInvalidOperation` error occurs.
  - Added information about handling errors when using the NFC Reader/Writer.
- 5.4. Tag Check Sequence
  - Noted that the `Mount()` and `MountRom()` functions can return `ResultInvalidFormatVersion`.
- 5.7. Tag Mounting Sequence
  - Noted that the `Mount()` and `MountRom()` functions can return `ResultInvalidFormatVersion`.
- 5.8. Tag Restoration Sequence
  - Noted that the `Restore()` function can return `ResultBackupError`.
- 5.10.2. Registration Information Retrieval Sequence
  - Modified the description so that owner registration and nickname registration are not required.
- 5.10.3. Application-Specific Region Access Start Sequence
  - Removed the requirement to notify the user if `ResultNeedCreate` is returned when accessing the application-specific region.
- 5.10.5. Application-Specific Region Check Sequence
  - Removed the text that states the `OpenApplicationArea()` function returns `ResultNotSupported`.
- 5.11.1. HOME Menu Transition Sequence
  - Noted that the NFP library must be finalized when transitioning to the HOME Menu.

- 5.11.2. Sleep Mode Transition Sequence
    - Removed unnecessary function calls.
- 5.11.3. Power Off Sequence
    - Added the transition to the POWER Menu to the figure.
    - Noted that tag detection must be stopped before transitioning to the POWER Menu.
- 6. Using the NFP Library
    - Added text indicating that `Finalize` must be called when transitioning to the HOME Menu or an applet.
    - Removed the explicit requirement to stop tag detection when transitioning to Sleep Mode.
- 6.2. Starting Tag Detection
    - Noted that the touch panel cannot be used during tag detection.
    - Added information about the tag recognition range (x, y: ±10 mm, z: +5 mm) for SNAKE and CLOSER.
- 6.4. Mounting a Tag
    - Revised the descriptions of the table tag state and the tag states of the return values of each function.
    - Noted that the `Mount()` and `MountRom()` functions can return `ResultInvalidFormatVersion`.
- 6.4.1. Restoring Tags
    - Noted that the `Restore()` function can return `ResultBackupError`.
- 6.5. Starting amiibo Settings
    - Updated the descriptions to note that a tag can no longer be initialized by jumping to **amiibo Settings**.
- 6.6.2.1. Creating the Application-Specific Region
    - Modified the sentence that recommends converting the data to write to big-endian.
- 6.10.1. Handling the HOME Button
    - Noted that the touch panel cannot be used during tag detection.
    - Noted that the NFP library must be finalized when transitioning to the HOME Menu.
- 6.10.2. Handling Sleep Mode Requests
    - Removed unnecessary function calls.
- 6.10.3. Handling the POWER Button
    - Noted that the touch panel cannot be used during tag detection.
- 6.10.4. Implementation Requirements for CTR
    - Modified the description to reflect the change in the method of checking the connection status.
    - Modified the times required for function calls.
    - Added a description of the timeout period for the connection status to become disconnected.
    - Added descriptions of function behavior when called on SNAKE.

- Removed the warning that recommended caching the connection status.
- Added text about how to update the status of the LED that indicates the remaining battery life of the NFC Reader/Writer.

## Version 1.0 (2014-10-15)

### Additions and Changes

- Initial version.

# 3DS Programming Manual: Dynamic Stereoscopy

Version 1.0

**Nintendo Confidential**

This document contains confidential and proprietary information of Nintendo, and is protected under confidentiality agreements as well as the intellectual property laws of the United States and of other countries. No part of this document may be released, distributed, transmitted, or reproduced in any form , including by any electronic or mechanical means and by including within information storage and retrieval systems, without written permission from Nintendo.

---

©2015 Nintendo. All rights reserved.

All company and product names in this document are the trademarks or registered trademarks of their respective companies.

CTR-06-0352-002-C

# 1. Introduction

This document explains what dynamic stereoscopy is and how to implement it. It starts by describing the theoretical framework that dynamic stereoscopy is built on.

It then details how to relax the theoretical constraints to ensure comfortable viewing and allow artistic control over rendering.

Finally, it describes the provided API.

## 1.1. Scope of This Document

This document deals only with perspective projections to generate images from 3D scenes. These perspective projections are well known to graphics developers. This document assumes that readers are already familiar with the perspective projection matrix.

## 1.2. Terminology Used in This Document

This document uses the following terminology.

- Real space
  The distance from the player to the upper LCD and other factors relevant to the stereoscopic display on CTR/SNAKE. To differentiate between these distances and the space in the application, this document uses the term *real space* to indicate the space that contains the player and the CTR/SNAKE system.
- Virtual space
  The space created in the application, as opposed to real space.
- Viewport
  The area in real space where the rendered images are displayed. This area lies in the plane of the screen. Most often, a game uses the entire upper screen as the viewport.
- Window
  The representation of the viewport in virtual space. It is a cross-section of the camera viewing volume that lies on the base plane/window plane. It is the window through which the user sees the virtual world when looking at the viewport. Objects lying on the window plane are seen by the user as being exactly at screen distance. Objects farther from the camera are seen as being behind the screen, and nearer objects are seen as being in front of the screen. The virtual screen is also defined as the representation of the screen in virtual space.
- Base camera
  The monoscopic camera that the application sets and creates according to the scene. This information is required in order to calculate cameras for the left eye and the right eye of the user.
- Eye camera
  A camera used to compute the image for the left or right eye.

**Figure 1-1. Virtual Space**

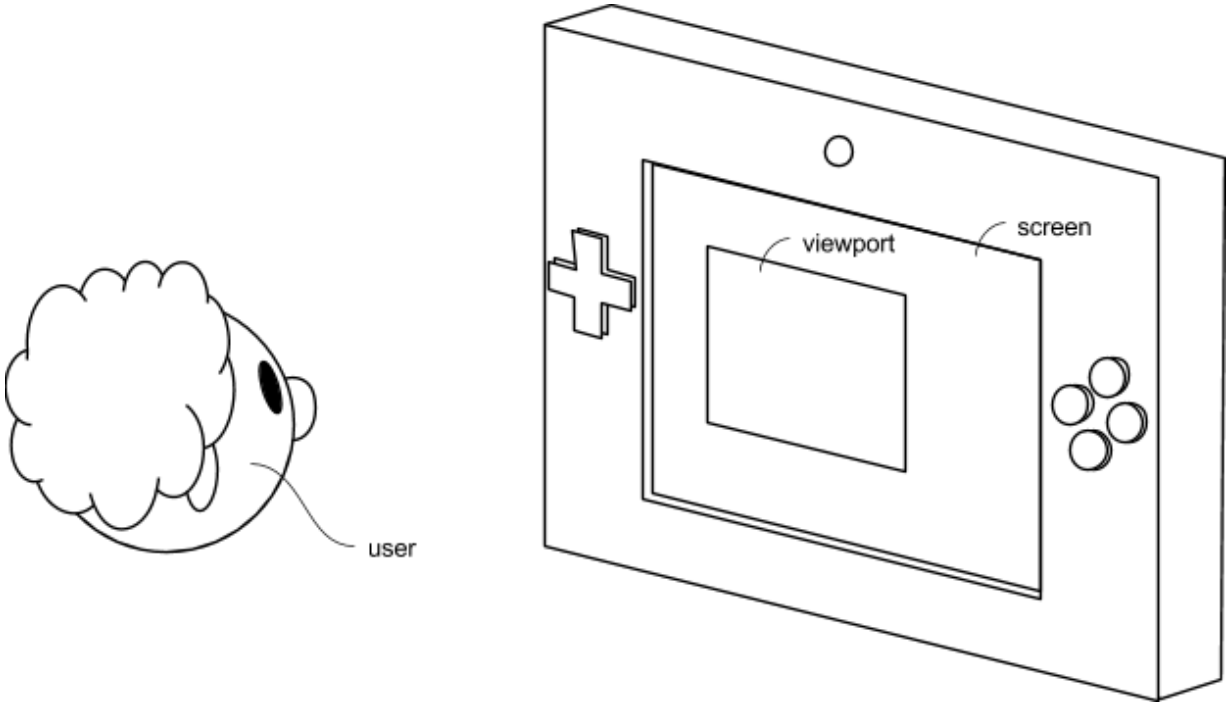
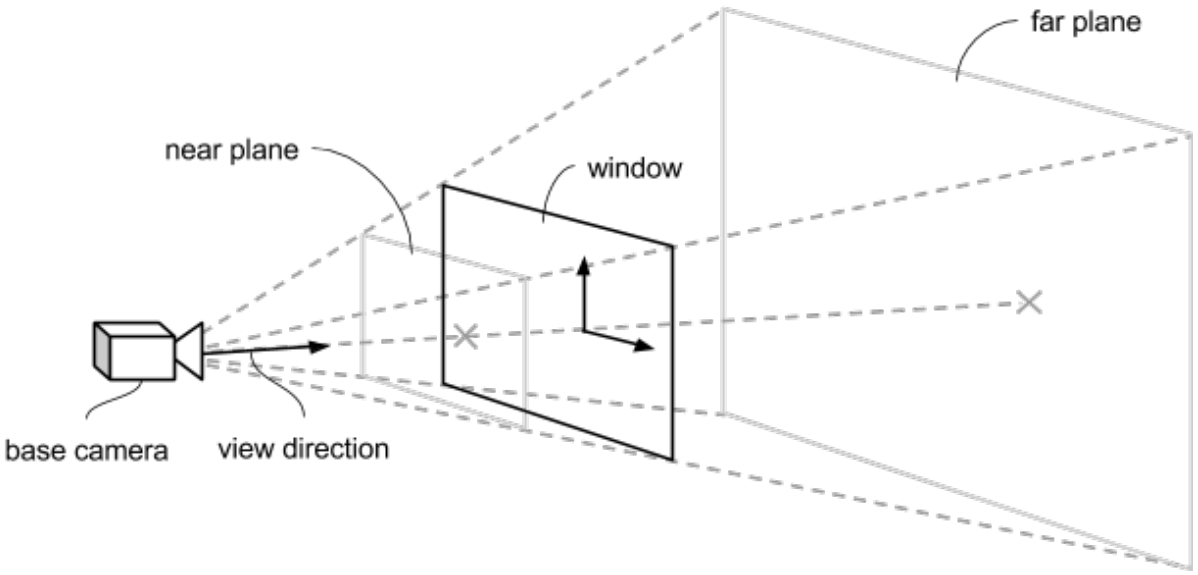



**Figure 1-2. Real Space**






# 2. Ideal Image Capture and Display

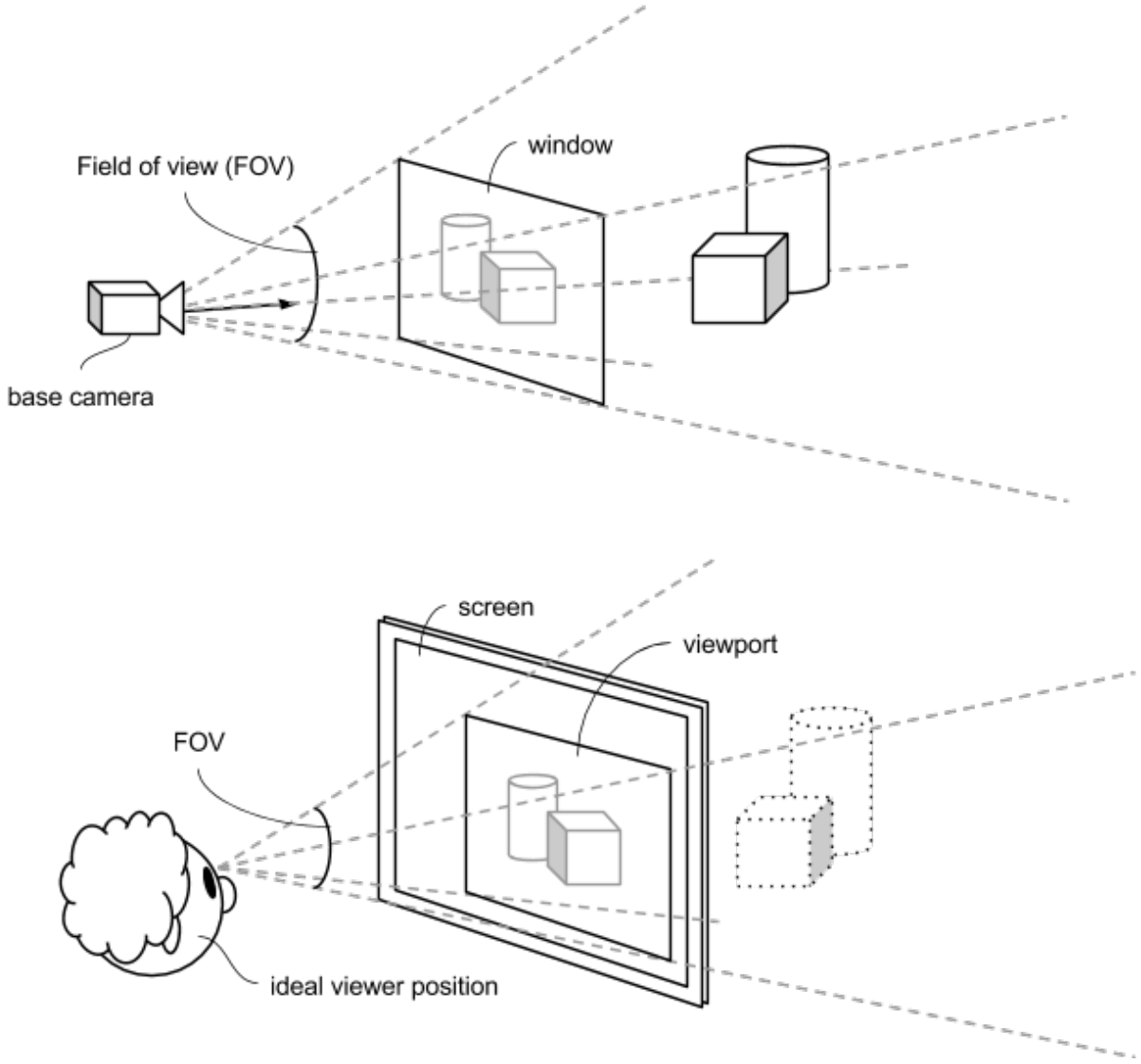
## 2.1. Ideal Monoscopic Effect

When computing an image from a 3D scene using a perspective matrix, a central projection is used to transform 3D points into 2D points. (See Figure 2-1.)

With a monoscopic effect, the projection plane can be placed at any distance from the center of the camera. The window content is rescaled to match the viewport size.

When a picture is displayed in a viewport, the ideal viewing position is at the point where the viewport has the same angular appearance as the window from the center of projection. The only degree of freedom for ideal display configuration is scale. Increasing the display size increases the ideal viewing position distance to the screen proportionally.

**Figure 2-1. Top: Capture Configuration; Bottom: Ideal Display Configuration**




The ideal viewing position is usually when the user is in front of the center of the screen, at a particular distance. This leads to the use of cameras that have a symmetric frustum, and where only field of view (FOV) is taken into account. However, it might be necessary to use an asymmetric frustum when the viewport does not cover the entire screen.

For the sake of simplicity, the following sections assume that the viewport covers the entire screen.

However, viewing a picture from a position that does not match the ideal viewing position does not

break the comprehension of the picture. In fact, this effect can be used artistically. For instance, cinema frequently uses different fields of view to convey different impressions to the audience. Some special effects can be performed by changing the field of view dynamically. (A common example is the dolly zoom.)

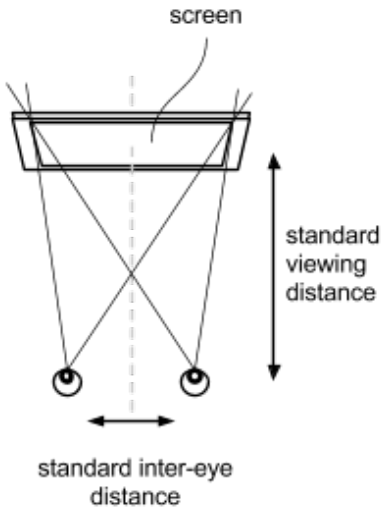
## 2.2. Ideal Stereoscopic Effect

3D viewing is performed by displaying two images on the same screen. One image is seen by the left eye, and the other by the right eye.

You can achieve an ideal stereoscopic effect by displaying the images such that the ideal viewing positions in virtual space match the positions of the eyes of the user in the standard viewing position.

The standard viewing position of the user is in front of the center of the screen. In this configuration there is horizontal symmetry. The standard eye positions are then positioned on each side of the center line. (See Figure 2-2.)

**Figure 2-2. Standard Viewing Configuration**



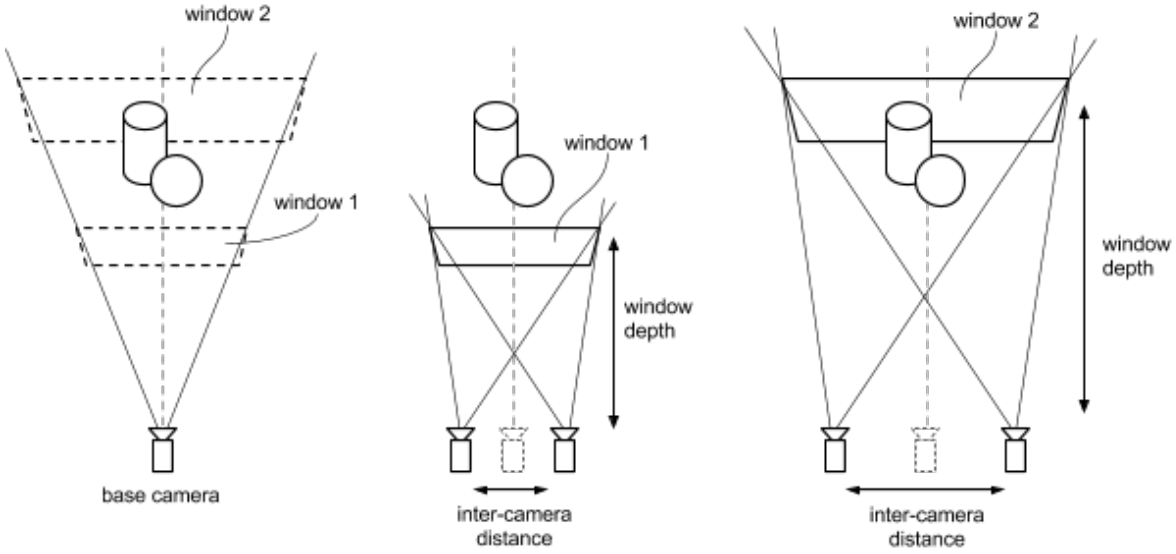

A stereoscopic camera comprises two base cameras that match the standard eye positions in virtual space.

An ideal stereoscopic camera is created by horizontally displacing two base cameras so that they match the eye positions and FOV of the user in the standard viewing position.

However, one degree of freedom remains. The virtual window can be placed at any distance from the center of the camera (see Figure 2-3). As the developer, you must determine the position of this window for stereoscopy. You must decide which plane in virtual space will be the plane of the screen, and which objects will appear on, behind, and in front of the screen. Changing the window depth (that is, its position) changes the apparent size of displayed objects.

**Figure 2-3. Left: Base Camera Configuration; Center: Stereoscopic Camera Configuration Using Window 1;**

**Right: Stereoscopic Camera Configuration Using Window 2**




The window can be set at any distance from the base camera. Consider window 1 and window 2 in Figure 2-3. In the stereoscopic camera configuration using window 1 (center), objects appear behind the screen. In the configuration using window 2 (right), objects appear in front of the screen.

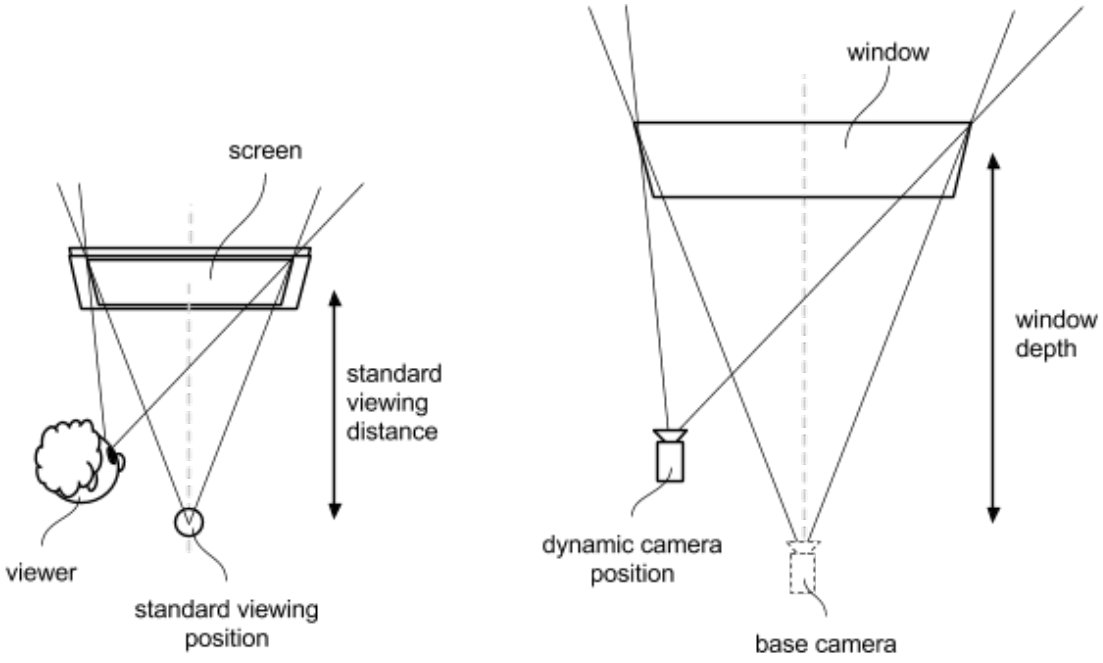
As shown in Figure 2-3, the windows for both cameras must coincide. As a result, the camera viewing directions (the window plane normals) must be the same, and their frustums must be made asymmetrical to encompass the window.

The FOV on the screen is determined by the standard viewing distance, based on the size of the screen. For ideal stereoscopy, have the camera use this FOV.

## 2.3. Ideal Dynamic Base Camera

If you can detect the user position relative to the screen, you can compute an image whose ideal position matches the current viewing position. This method is called a *dynamic perspective*.

**Figure 2-4. Dynamic Perspective**




Assume that you know the exact position of the user in the viewport frame of reference. To position the camera, you must define the window frame of reference in the virtual world. You can derive the window from the base camera that corresponds to the standard viewing position in real space.

You can position the window on any plane orthogonal to the viewing direction, at any positive distance. The window is the positional center of rotation, so the appearance of rendered objects changes depending on the distance from the camera to the window. To achieve an ideal dynamic camera, you must rotate the base camera, such as to the position of the dynamic camera in Figure 2-4. The same is true for a stereoscopic camera.

## 2.4. Ideal Dynamic Stereoscopic Camera

To develop a dynamic stereoscopic camera, define the position of the window in the virtual world, and use the positions of both user eyes to position cameras, as was the case for dynamic base cameras.

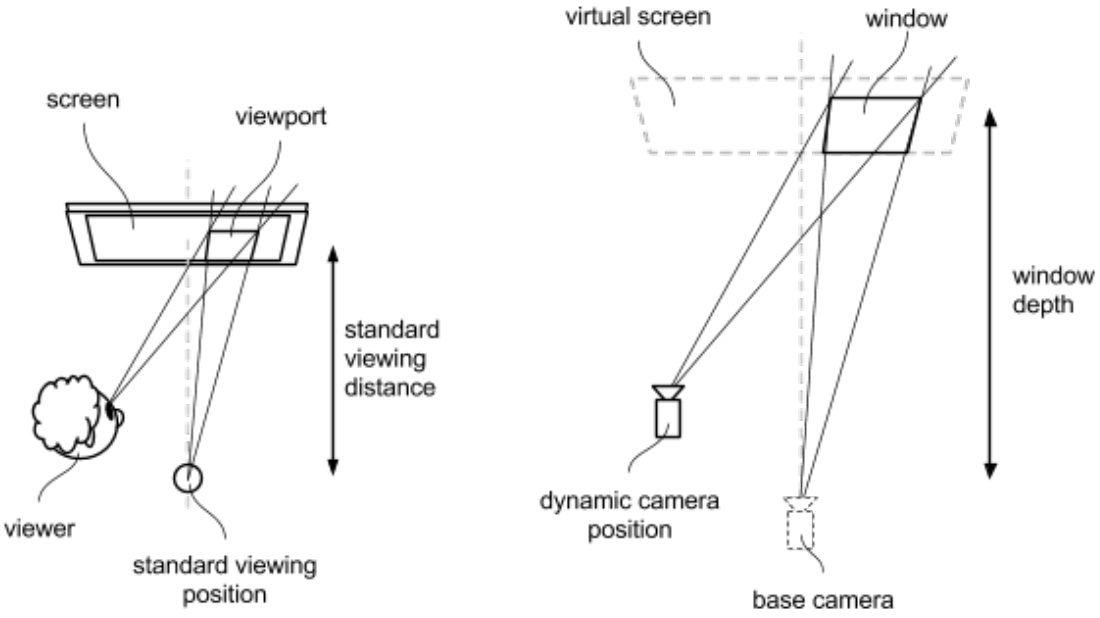
## 2.5. Using a Viewport Not Covering the Entire Screen

The operations described to this point assume that a picture is displayed on the entire screen. However, displaying a 3D picture in a sub-area needs special care.

The system can only know the relative position of the user to the screen. If the developer plans to

display a 3D picture in a sub-area of the screen, the viewport and the position of the user relative to the viewport must be expressed to compute the correct camera position. To do so, the developer must specify the exact position of the viewport in the screen to the ideal dynamic stereo camera.

**Figure 2-5. Dynamic Camera and Viewport**






# 3. Integration in SNAKE

## 3.1. Face Tracking System Limitations

### 3.1.1. Estimating Depth

So far the position of the user relative to the screen is assumed to be known. However, that is not the case with SNAKE. The system's face tracking feature can determine the user's angle to the screen with good precision, but it cannot compute the user's distance from the screen.

Instead, the system assumes that the user is located at the optimal distance, which is the distance at which the 3D effect is best perceived.

### 3.1.2. Using an Average For Distance Between Eyes

Even if the position of both eyes can be tracked by the system, the real distance between the eyes for a particular user is not known. The apparent distance between the eyes in the image retrieved from the camera varies depending on the user's distance from the screen, so it cannot be used for the actual distance between the eyes. A virtual user with an average distance between eyes is assumed. An average value is assumed for the distance between eyes for a user positioned at the optimal distance from the screen. The same assumption is used for pictures of the user retrieved from the camera. The eye positions of this virtual user are used for all computations instead of the raw ones.

### 3.1.3. Handling User Detection Failure

Sometimes, the user might move out of the field of view of the tracking system. Other times, poor conditions might make recognition impossible. In these situations, the system calculates an estimate for the user’s position, based on prior position information and input from the gyro sensors. The estimate begins with the last-detected position of the user. If there is no gyro sensor input for some time—implying that the SNAKE system is not moving—the estimated position is shifted over time toward the center of the screen. If the tracking system remains unable to detect the user, it assumes that the user has returned to a position facing the center of the screen. If the SNAKE system is moving, the calculations assume that the user is static, and estimate the user's position relative to the moving console.

This behavior has been designed to provide a user experience at least as good as with 3D screens that do not support Super-Stable 3D. In addition, it provides a good user experience when the user moves the device.

## 3.2. Comfort and Maximum Parallax

Experiments have shown the importance of maintaining parallax within limits. There is a limit distance for parallax, and this value must be taken into consideration when setting the distance for an object in the images presented to the left and right eyes. If this limit is exceeded, most users will feel uncomfortable.

Using the ideal stereoscopic camera described in the previous section, this can happen when the object is situated very far from the window plane. If an object located infinitely far from the user is rendered at a distance equal to the distance between eyes, the parallax is very large. The following methods can be used to ensure that this constraint is enforced.

### 3.2.1. Near and Far Planes

Setting near planes and far planes nearer to the window plane reduces the maximum parallax that a visible object can have.

### 3.2.2. Reducing Distance Between Eyes

Another way to reduce maximum parallax is to render the scene for a reduced distance between eyes. This reduces the parallax on the farthest objects. However, it also reduces the 3D effect on the nearest objects.

### 3.2.3. Increasing FOV

As shown in the following section, parallax in the far plane can also be reduced by increasing the FOV.



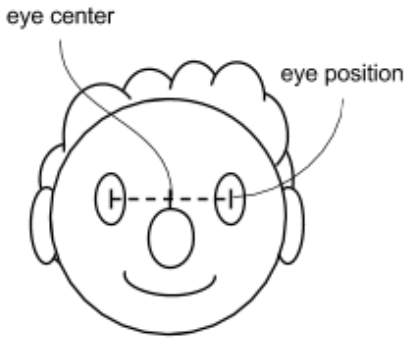
# 4. Dynamic Stereoscopic Camera Settings

With the previously defined theoretical model in mind, you can relax some constraints to enable some control over the camera for artistic, comfort, or technical reasons.

To implement the following features, you must consider the position of the user's eyes in two different ways: in terms of the absolute position of the user, and in terms of the relative positions of the eyes. This enables you to understand stereoscopic effects and dynamic perspective effects independently.

For dynamic perspective effects, the center point between the eyes is used as the user's position. For stereoscopic effects, the positions of the eyes relative to this center point are used.

**Figure 4-1. Decoupling Stereoscopic Effects From Dynamic Perspective Effects**

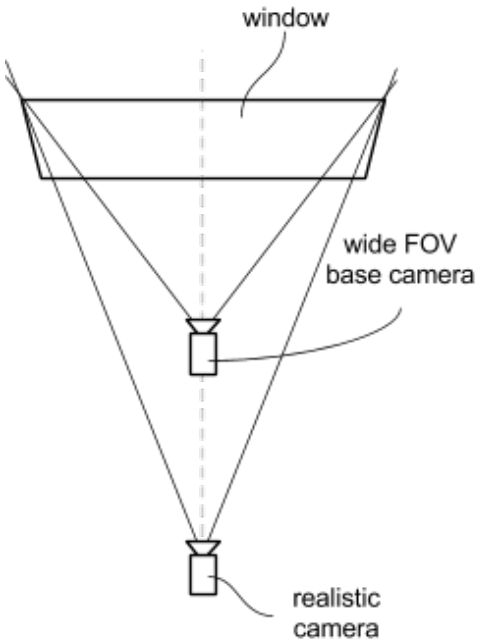



## 4.1. Wide FOV

When dealing with a base camera, using a specific FOV is natural for game developers. However, ideal stereoscopy needs a fixed FOV—a standard FOV—that matches the angular appearance of the screen as seen by the user.

Most of the time, these specific FOVs are larger than the standard FOV. These larger FOVs are called *wide FOVs*.

**Figure 4-2. Wide-FOV Camera**



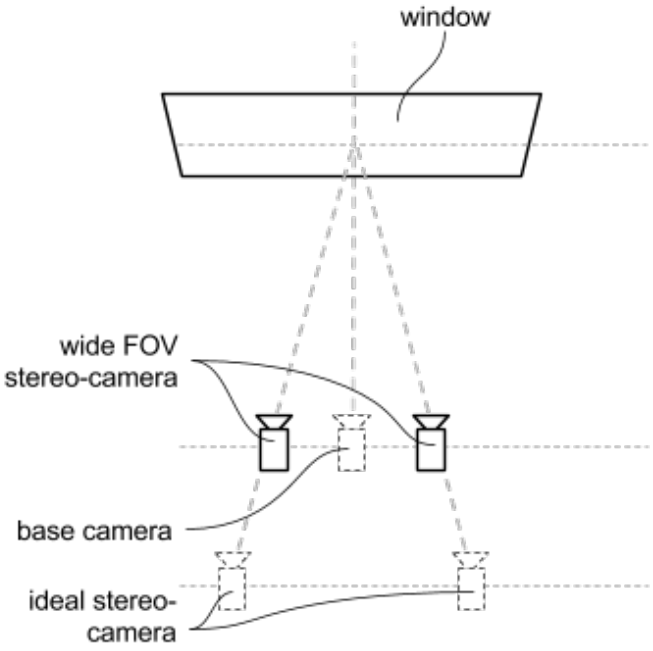

It is possible to compute a realistic camera that matches the standard FOV and whose window matches the window of the base camera. However, using another FOV may be needed for artistic reasons.

If you want to use a wide FOV, you can reduce the deformation of objects displayed in the center of the screen by adeptly setting the distance between the left and right cameras. To minimize these

deformations, match the direction at which both cameras are pointing toward the center of the window to the direction at which the user is looking toward the center of the screen.

As shown in Figure 4-3, deformation is minimized by setting the inter-camera distance for the ideal stereo cameras and the distance between the left and right wide FOV cameras proportional to their distances from the window.

**Figure 4-3. Ideal Inter-Camera Distance for Wide-FOV Cameras**




When using dynamic perspective, you need to consider methods for computing the camera movement from the user's position. Consider the following methods.

The first method scales the movement as the inter-eye distance was scaled. The effect of this method is that an object at the center of the screen is viewed from the appropriate angle. However, far objects seem to move less than is normal.

The second method reproduces the movement of the user on the cameras without scaling. This method gives the impression that far objects move realistically, while near objects seem to rotate or deform.

Both methods can be justified, and intermediate behaviors can be useful depending on the scene. Any intermediate behavior can be described by a scalar factor between 0 and 1, the wide FOV motion factor.

**Figure 4-4. FOV Motion Factor Set to 0**

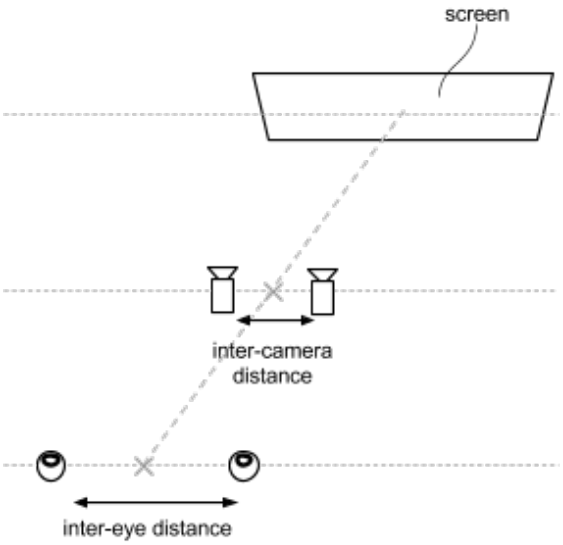
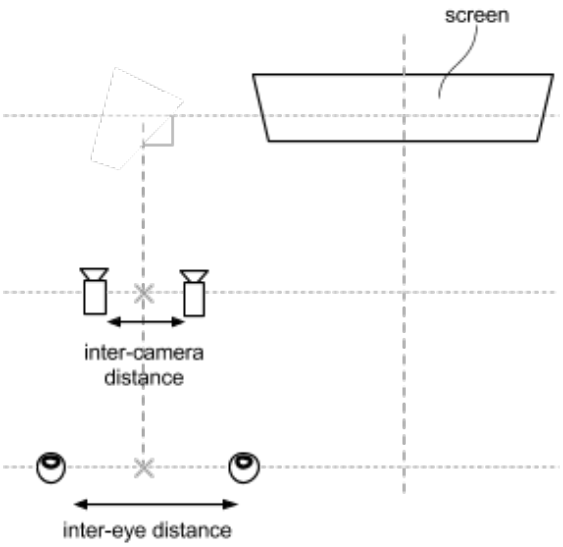


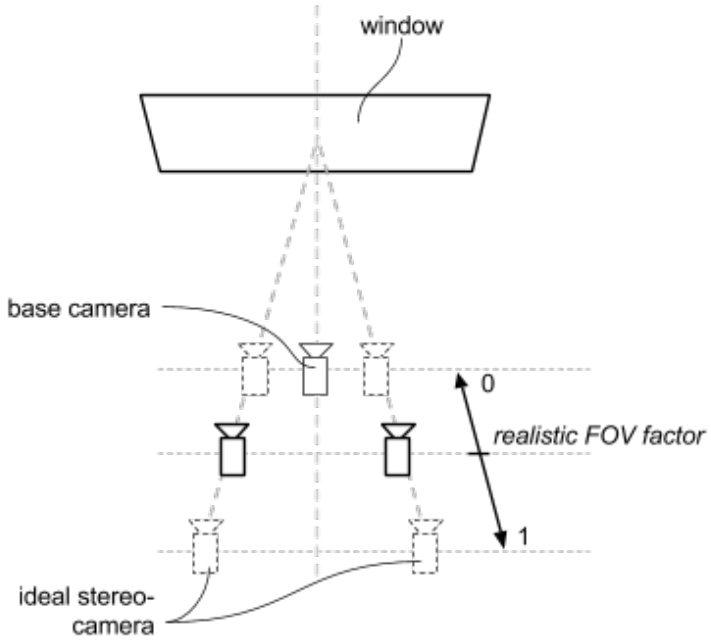

**Figure 4-5. FOV Motion Factor Set to 1**




## 4.2. Realistic FOV Factor

If you want to smoothly transform a wide FOV stereoscopic camera to an ideal stereoscopic camera, try interpolating between them for an interesting result. Use linear interpolation to move the virtual camera from one position to another. This linear interpolation factor is the realistic FOV factor.

**Figure 4-6. Realistic FOV Factor**




## 4.3. Dynamic Perspective Amplitude

You can amplify or reduce a dynamic perspective effect by applying a multiplicative factor. This dynamic perspective amplitude factor corresponds to the user’s displacement from the standard position.

You can modify this amplitude independently on each axis depending on the desired effect or specific constraints.

## 4.4. Dynamic Perspective Clamping

The area where the user can be detected by the system is limited by the field of view of the camera. When the user leaves this area, the system loses the user's position. To provide visual feedback to the user before leaving this area, you can clamp the maximum viewing angle to a narrower one.

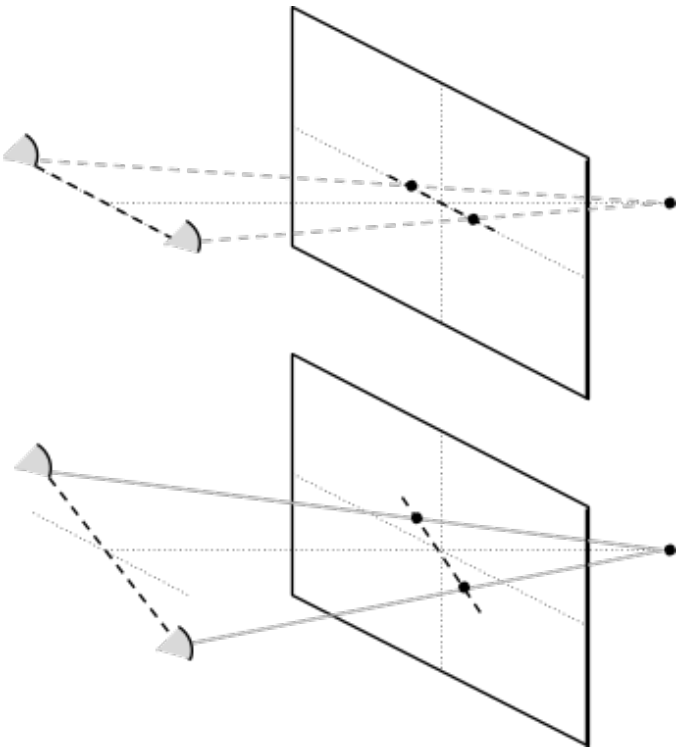
This clamping can also be used to control the volume of space that can be seen by the camera.

## 4.5. Tilt and Improved Static Stereoscopic Camera

If camera displacement is deactivated by setting either the clamping factor or the amplitude factor to `0`, the camera works like a static stereoscopic camera.

Even so, the angle between the user and the console can be determined from the detected position of the eyes, and this tilt information can be applied to the virtual cameras. This can improve stereoscopic quality when the user tilts the screen as if turning a steering wheel.

**Figure 4-7. Improved Static Stereoscopic Camera**



However, in rare instances using tilt this way can cause problems. Also, moving the virtual camera can generate artifacts. If there are problems, disable the tilt so that the camera remains horizontal.

## 4.6. Parallax Distance Limit

To guarantee the comfort of the user, the developer can specify a maximum parallax distance. A maximum inter-camera distance is derived from this constraint, such that an object on the far plane would have at most this parallax distance. However, this constraint is not enforced for objects on the near plane.

## 4.7. 3D Slider and Stereo Factor

For users to tune the 3D effect to their liking, the inter-camera distance must be adjusted using the 3D slider. The developer can provide a stereo factor that takes values in the range from 0 through 1. The inter-camera distance can be shortened by multiplying the 3D slider value by this factor. You can use this technique to reduce the 3D effect if necessary, or to perform smooth transitions between 2D and 3D images.

CONFIDENTIAL

# 5. Notes for Integration

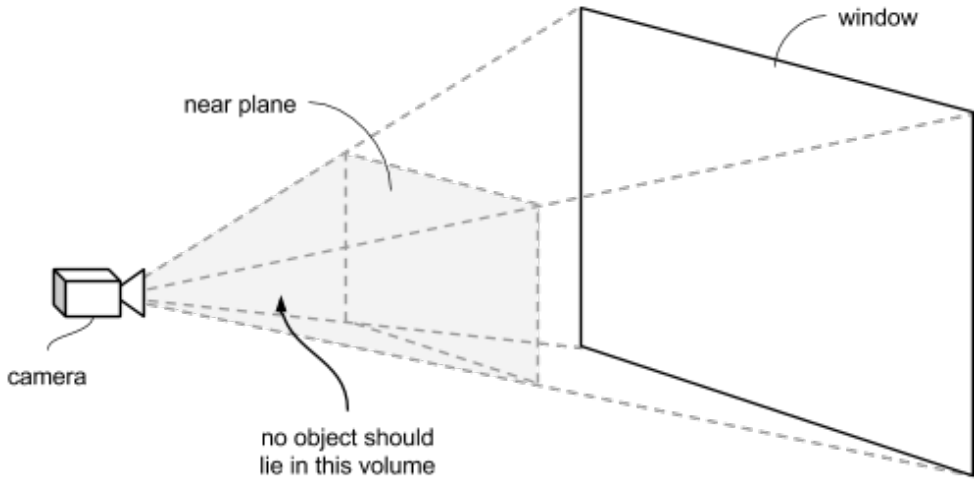
## 5.1. Determining Camera Occupancy

Camera placement is constrained to a set of acceptable positions in the game. It is desirable to position the camera so that it is out of the way of any geometry objects. When using a dynamic stereoscopic camera, you want to accurately reproduce the movements of the user. For this reason, the camera cannot be placed in locations where it would be possible to locate a static base camera but not the final eye cameras. This placement would lead to results that do match the movement of the player.

Take these constraints into consideration before deriving the two eye cameras from the base camera. To avoid collisions with geometry objects, you must secure regions for the camera that are out of the way.

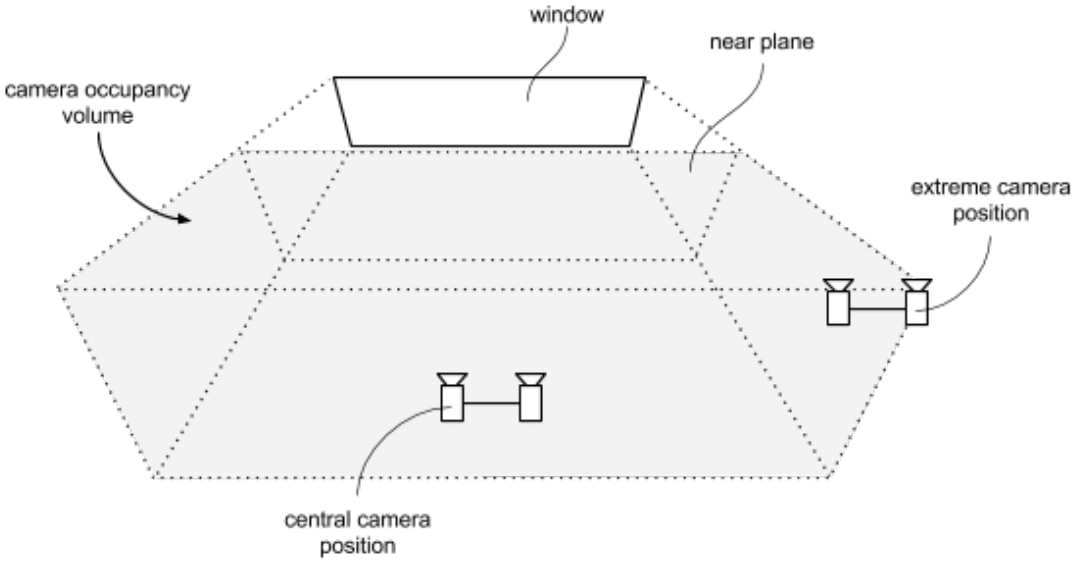
Some particular camera positions are acceptable. In these positions, there are no geometry objects in the camera FOV between the camera and the near plane. Otherwise, rendering produces artifacts.

**Figure 5-1. Camera Occupancy for a Static Base Camera**



For a dynamic stereoscopic camera, you must confirm that a particular camera is in an acceptable position for all possible user positions. For this purpose, the camera occupancy volume can be defined as the frustum from the camera plane to the near plane. The sides of this volume are the planes that stretch from the extreme reachable positions of the camera to the edges of the window. See Figure 5-2.

**Figure 5-2. Camera Occupancy for a Dynamic Stereoscopic Camera**






# 6. DynamicStereoCamera Class

## 6.1. Specifying the Window

When you are using a static camera, you specify the frustum. But when you are using a dynamic camera, you need to specify the window. You must also specify the near and far planes.

### 6.1.1. Setting the Window From Its Size in Virtual Space

You can specify the window width and the depth at which it is placed from the center of the camera. The resulting window is centered on the z-axis of the camera, and its height is set to match the aspect ratio of the device's upper screen.

Note that the ratio of the window width and the window depth determines the FOV.

This method assumes that the resulting image will be displayed on the entire upper screen.

**Code 6-1. SetWindowFromSize Function Definition**

```
void SetWindowFromSize(const f32 windowWidth,
                       const f32 nearDepth,
                       const f32 windowDepth,
                       const f32 farDepth);
```

## 6.1.2. Deriving the Window From the Base Camera Frustum

You can give left, right, bottom, and top tangent angles of the base camera frustum. The window is then defined as the intersection of this frustum and the plane at window depth. You can use this method to describe viewports that do not cover the entire screen. Compute the viewport border tangents from a standard viewing position in front of the center of the screen. It is assumed that the resulting image will be displayed on the largest area of the screen that matches the viewport description.

**Code 6-2. SetWindowFromTangent Function Definition**

```
void SetWindowFromTangent(const f32 tangentLeft,
                          const f32 tangentRight,
                          const f32 tangentBottom,
                          const f32 tangentTop,
                          const f32 nearDepth,
                          const f32 windowDepth,
                          const f32 farDepth);
```

## 6.1.3. Setting the Window From the Viewport

To deal with a viewport that only covers a portion of the screen, you need to know the location of the viewport within the screen. The following member function is useful when you are using this kind of viewport and a realistic FOV.

**Code 6-3. SetWindowFromViewport Function Definition**

```
void SetWindowFromViewport(const f32 screenHorizontalHalfFOVTangent,
                           const f32 viewportLeftInUnitSceen,
                           const f32 viewportRightInUnitSceen,
                           const f32 viewportBottomInUnitSceen,
                           const f32 viewportTopInUnitSceen,
                           const f32 nearDepth,
                           const f32 windowDepth,
                           const f32 farDepth);
```

### 6.1.4. Setting the Window Based on an Existing Projection Matrix

You can also set the window by specifying the window depth and the projection matrix of the base camera used to calculate the frustum.

However, this method can only be used for a projection matrix that targets the entire upper screen and assumes a symmetric frustum.

**Code 6-4. SetWindowFromProjectionMatrix Function Definition**

```
void SetWindowFromProjectionMatrix(const nn::math::Matrix44 *proj,
                                   const f32 windowDepth);
```

## 6.2. Specifying the Base Camera's View

**Code 6-5. SetView Function Definition**

```
void SetView(const nn::math::MTX34* viewMatrix);
void SetView(const nn::math::VEC3& position,
             const nn::math::VEC3& upDirection,
             const nn::math::VEC3& forwardDirection);
```

Either specify the view matrix created by a function such as `nn::math::MTX34LookAt`, or directly specify the parameters for the base camera's position. Make sure that you use the right-handed coordinate system adopted by the CTR graphics system when specifying the view matrix or the `nn::math::Vector3` arguments.

## 6.3. Computing Left and Right Cameras

**Code 6-6. ComputeViewsAndProjections Function Definition**

```
f32 ComputeViewsAndProjections(
    nn::math::Matrix44* projL, nn::math::Matrix34* viewL,
    nn::math::Matrix44* projR, nn::math::Matrix34* viewR,
    const nn::math::VEC2* leftEyeTangent,
    const nn::math::VEC2* rightEyeTangent,
    const Setting& setting,
    const nn::math::PivotDirection pivot = nn::math::PIVOT_UPSIDE_TO_TOP
) const;
```

This method is used to compute projection and view matrices to render left and right images for the left and right eyes.

This method assumes that the user is placed at the optimal distance from the surface of the LCD screen. It can provide a realistic stereoscopic view and other non-realistic (or artistic) views based on the *setting* parameter, explained below. This method can take into account the user's head position, and adjust the dynamic perspective accordingly.

The *leftEyeTangent* and *rightEyeTangent* parameters describe the position of the left and right eyes of the user relative to the screen. These values can be retrieved by a call to the `nn::qtm::CTR::GetTrackingData()` function.

When either *leftEyeTangent* or *rightEyeTangent* is NULL, this method produces the same behavior as a static stereoscopic camera. Set these parameters to NULL when developing for the CTR platform.

The upward direction of the camera in the generated projection matrix is the upward direction as determined by the *pivot* parameter. If *pivot* is set to `nn::math::PIVOT_NONE`, no rotation is applied. If an invalid value is specified for *pivot*, the function halts on an assertion. The default value is `nn::math::PIVOT_UPSIDE_TO_TOP`.

The function returns the factor for the distance between cameras, $f_I$. This factor is described in the appendix.

## 6.4. Setting Class

**Code 6-7. Setting Class Definition**

```
struct Setting
{
    f32 StereoFactor;
    f32 ParallaxDistanceLimit;
    f32 DynamicPerspectiveAmplitudeX;
    f32 DynamicPerspectiveAmplitudeY;
    f32 DynamicPerspectiveClampingLeft;
    f32 DynamicPerspectiveClampingRight;
    f32 DynamicPerspectiveClampingBottom;
    f32 DynamicPerspectiveClampingTop;
    f32 RealisticFovFactor;
    f32 WideFovMotionFactor;
    bool TiltEnabled;
}
```

You can use the `Setting` class passed as an argument to the `ComputeViewsAndProjections()` function to adjust the behavior of the stereoscopic camera.

- `StereoFactor` adjusts the stereoscopic effect, as described in 4.7 3D Slider and Stereo Factor.
- `ParallaxDistanceLimit` specifies the parallax distance limit, in millimeters. This limit is the maximum distance between the object in the image for the left eye and the object in the image for

the right eye. For more information, see 4.6 Parallax Distance Limit.

- `DynamicPerspectiveAmplitudeX` and `DynamicPerspectiveAmplitudeY` are multiplication factors for the user position. They amplify or reduce the dynamic perspective effect for a head movement.
- `DynamicPerspectiveClampingLeft`, `DynamicPerspectiveClampingRight`, `DynamicPerspectiveClampingBottom`, and `DynamicPerspectiveClampingTop` are used to clamp the user's position horizontally and vertically. They take values in the range from 0 through 1. When set to 1, the position is clamped to the limit value obtained using `GetTrackingData`. If set to a value in the middle of the range, the position is clamped to the corresponding fraction of the limit value. If set to `0`, the camera stops moving in that direction. Note that the position is clamped before the amplitude set by `DynamicPerspectiveAmplitude` is applied.
- `RealisticFovFactor` is used to interpolate the camera position between the realistic camera and the base camera, as described in 4.2 Realistic FOV Factor.
- `WideFovMotionFactor` adjusts how the camera moves in response to movements by the user when a wide FOV is being used, as described in 4.1 Wide FOV.
- `TiltEnabled` indicates whether tilt is enabled or disabled. For more information, see 4.5. Tilt and Improved Static Stereoscopic Camera.

## 6.5. Occupancy Volume

**Code 6-8. OccupancyVolume Structure Definition**

```
struct OccupancyVolume
(
    math::VEC3 CameraCenter;
    math::VEC3 ForwardDirection;
    math::VEC3 OrthoUpDirection;
    math::VEC3 RightDirection;
    f32 NearPlaneDepth;
    f32 NearLeft;
    f32 NearRight;
    f32 NearBottom;
    f32 NearTop;
    f32 CameraLeft;
    f32 CameraRight;
    f32 CameraBottom;
    f32 CameraTop;
};
```

Use the `OccupancyVolume` structure to express the occupancy volume of the camera. To guarantee that artifacts do not appear unnatural due to the camera crossing paths with objects, make sure that no objects are located inside this volume.

- `CameraCenter` is the camera position when dynamic perspective is disabled. If

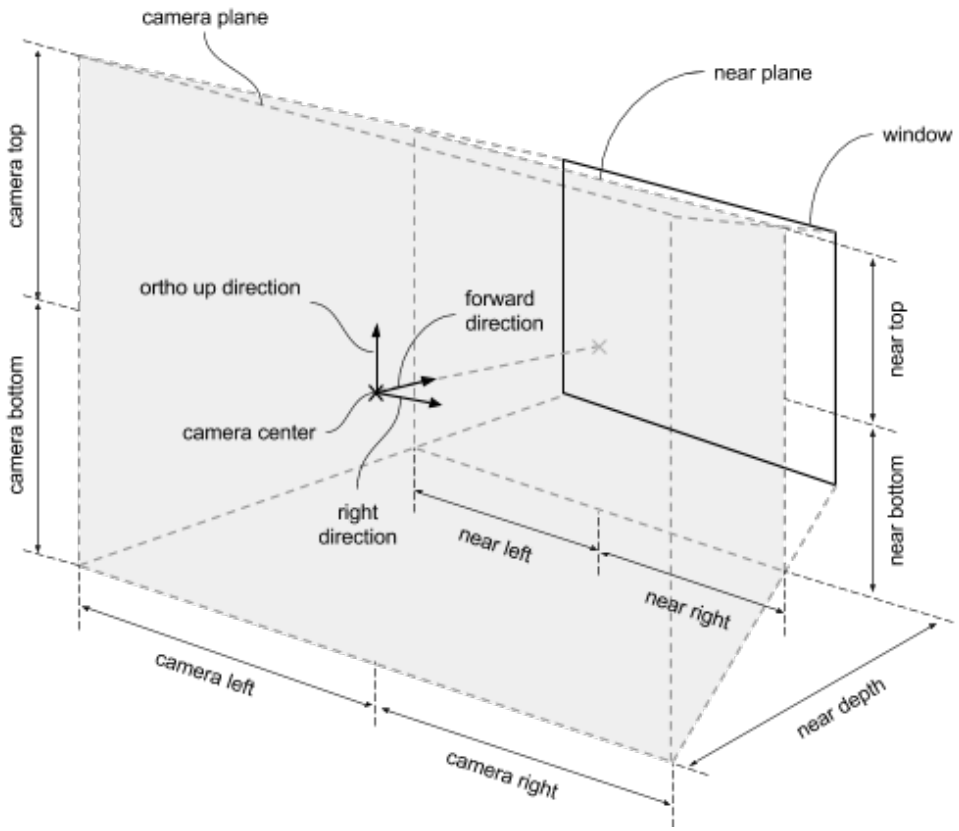
`RealisticFovFactor` is set to `0`, this position is the same as that specified by the `SetView` member function. For any value greater than 0, the camera position is adjusted along the axis in the forward direction, approaching the realistic FOV. The plane on which the camera moves for dynamic perspective is called the *camera plane*.

- `ForwardDirection` is the unit vector in the forward direction. The value is the same as that set by the `SetView` member function.
- `OrthoUpDirection` is the unit vector in the upward direction. It bisects at right angles with `ForwardDirection`.
- `RightDirection` is the unit vector in the rightward direction. Because it bisects at right angles with `ForwardDirection` and `OrthoUpDirection`, you can calculate the rightward vector from their cross product.

$$RightDirection = ForwardDirection \times OrthoUpDirection$$

- `NearPlaneDepth` is the distance from the camera position to the near plane.
- `NearLeft` and `NearRight` define the range of the occupancy volume in the horizontal direction (in the direction of the rightward vector) in the near plane.
- `NearBottom` and `NearTop` define the range of the occupancy volume in the vertical direction (in the direction of the upward vector) in the near plane.
- `CameraLeft` and `CameraRight` define the range of the occupancy volume in the horizontal direction in the camera plane.
- `CameraBottom` and `CameraTop` define the range of the occupancy volume in the vertical direction in the camera plane.

**Figure 6-1. Occupancy Volume Parameters**






# 7. Appendix: Formulas

## 7.1. Constants

The current system uses the following values.

$I_{eye}$ The standard distance between user eyes: 62 mm.

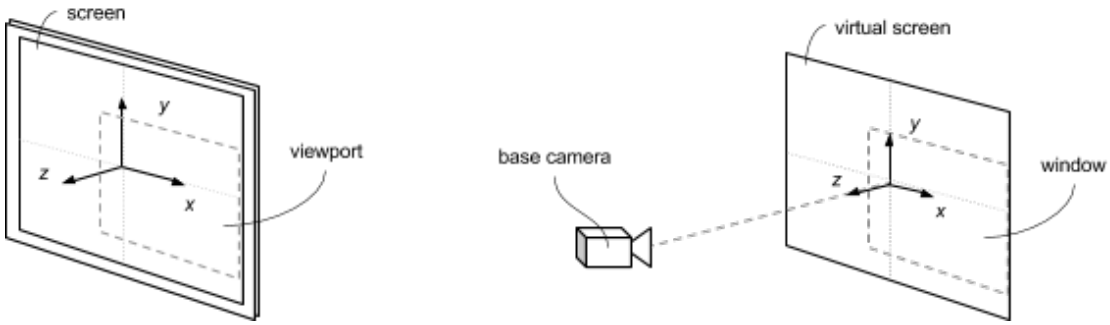
$d_{ideal}$ The standard distance from the surface of the upper screen to the user: 316.5 mm (the average of SNAKE (301 mm) and CLOSER (332 mm)).

$W_{screen}$ The screen width: 95.4 mm (the average of SNAKE (84.6 mm) and CLOSER (106.2 mm)).

$H_{camera}$ The distance from the center of the screen to the center of the camera: 37.62 mm (the average of SNAKE (34.38 mm) and CLOSER (40.86 mm)).

# 7.2. Converting Between Virtual and Real Spaces

**Figure 7-1. Left: Real Space Frame of Reference; Right: Virtual Space Frame of Reference**




The frames of reference in real space and virtual space were defined as shown in Figure 7-1. In both frames of reference, the origin is the center point of the screen.

Assume that the orthogonal projection of the camera position on the window is the center of the window.

The following scale is used for converting the distance from real coordinates to virtual coordinates.

$$scale_{r2v} = \frac{W_{window}}{W_{viewport}}$$

$W_{window}$ is the width of the window in virtual space,

$W_{viewport}$ is the width of the corresponding viewport in real space.

# 7.3. User Position

$X_{lefteye}$ and $X_{righteye}$ are defined as the positions of the user's eyes in real space. These positions are only known by their angular tangents.

$$(u_{left\ eye}, v_{left\ eye})$$

$$(u_{right\ eye}, v_{right\ eye})$$

Assuming that the user is placed at optimal distance, where the 3D effect is best perceived, the user's eye positions can be extrapolated.

$$X_{left\ eye} = (u_{left\ eye} \cdot d_{ideal}, v_{left\ eye} \cdot d_{ideal} + scale_{r2v} \cdot H_{camera}, d_{ideal})$$

$$X_{right\ eye} = (u_{right\ eye} \cdot d_{ideal}, v_{right\ eye} \cdot d_{ideal} + scale_{r2v} \cdot H_{camera}, d_{ideal})$$

These positions can be broken down into a user position $X_{user}$ and an inter-eye vector $\Delta_{eyes}$ (the vector from the left eye to the right eye).

$$X_{user} = \frac{1}{2}(X_{left\ eye} + X_{right\ eye})$$

$$\Delta_{eyes} = X_{right\ eye} - X_{left\ eye}$$

The unit vector in the direction of $\Delta_{eyes}$ is called $\delta_{eyes}$.

$$\delta_{eyes} = \frac{\Delta_{eyes}}{||\Delta_{eyes}||}$$

The position of the user is also represented as a displacement $X'_{user}$ from the standard position.

$$X'_{user} = X_{user} - (0,0,d_{ideal})$$

## 7.4. Computing the Distance Between Cameras

$I$ is the distance between the cameras for the left and right images.

$$I = ||X_{right\ cam} - X_{left\ cam}||$$

This distance is equal to the realistic inter-camera distance $I_{realistic}$ when configured for realism. It can be computed from the distance between eyes, $I_{eye}$.

$$I_{realistic} = scale_{r2v} \cdot I_{eye} \cdot I_{amp}$$

### 7.4.1. Computing Wide FOV

If the camera distance is not at the ideal distance (wide FOV configuration), the distance between cameras must be scaled down with the factor.

$$k_{fov} = \frac{d_{window}}{scale_{r2v} \cdot d_{ideal}}$$

$d_{window}$ is the distance between the window and the camera, computed as an interpolation between the distance specified by the developer, $d_{window\ base}$, and the standard distance $d_{ideal}$ in virtual space.

$$d_{window} = lerp(d_{window\ base}, scale_{r2v} \cdot d_{ideal}, f_{realistic})$$

$f_{realistic}$ is the realistic FOV factor.

### 7.4.2. Computing the Parallax Distance Limit

To guarantee the comfort of the user, the developer can specify a parallax distance limit $P_{limit}$ and ensure that this limit is not exceeded. A maximum inter-camera distance $I_{max}$ is derived from the following constraints.

$$I_{max} = \frac{d_{far}}{d_{far} - d_{window}} \cdot scale_{r2v} \cdot P_{limit}$$

$d_{far}$ is the depth of the far plane, derived from the value of $d_{far\ base}$ specified by the developer.

$$d_{far} = d_{far\ base} + d_{window} - d_{window\ base}$$

### 7.4.3. Inter-Camera Distance Factor

The final inter-camera distance $I$ is used for rendering.

$$I = f_{3D} \cdot f_{stereo} \cdot min(k_{fov} \cdot I_{realistic}, I_{max})$$

The inter-camera distance factor $f_I$, which represents how much the realistic distance between cameras has been scaled down, can be derived from $I$.

$$f_I = \frac{I_{realistic}}{I}$$

## 7.5. Computing the Central Camera Position

Let $X_{leftcam}$ and $X_{rightcam}$ be the positions of the cameras for the left and right eyes in the virtual space frame of reference. The center point between the two cameras is defined as $X_{cam}$.

$$X_{cam} = \frac{1}{2}(X_{left\ cam} + X_{right\ cam})$$

The reference camera position, $X_{camref}$, lies on the $z$-axis, at the standard distance in virtual space.

$$X_{cam\ ref} = (0,0,d_{window})$$

$X_{cam}$ is obtained by calculating the user's displacement to this reference camera position.

$$X_{cam} = X_{cam\,ref} + f_{amplitude} \cdot k_{fov\,motion} \cdot scale_{r2v} \cdot clamp(X'_{user})$$

$f_{amplitude}$ is the dynamic perspective amplitude factor.

$k_{fov\ motion}$ is the amplitude factor computed from the wide FOV motion factor.

$$k_{fov\,motion} = lerp(k_{fov}, 1, k_{wide})$$

$clamp(X'_{user})$ is the user displacement, clamped to the maximum detectable user displacement.

## 7.6. Computing Final Eye Camera Positions

The final positions of the eye cameras are computed by displacing both cameras from the central camera position. This computation can include or exclude the tilt parameter.

### 7.6.1. When Tilt Is Enabled

$$X_{left\,cam} = X_{cam} - \frac{1}{2} \cdot I \cdot \delta_{eyes}$$

$$X_{right\,cam} = X_{cam} + \frac{1}{2} \cdot I \cdot \delta_{eyes}$$

### 7.6.2. When Tilt Is Disabled

$$X_{left\,cam} = X_{cam} - \frac{1}{2} \cdot I \cdot (1,0,0)$$

$$X_{right\,cam} = X_{cam} + \frac{1}{2} \cdot I \cdot (1,0,0)$$

# Revision History

## Version 1.0 (2015-01-15)

### Additions and Changes

- Initial version.